ANALYSIS AND DESIGN OF ADVANCED CACHING SOLUTIONS FOR THE MODERN WEB

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Qi Huang

August 2014

ANALYSIS AND DESIGN OF ADVANCED CACHING SOLUTIONS FOR THE MODERN WEB

Qi Huang, Ph.D.

Cornell University 2014

The rise of modern Web applications has seen a surge in the quantity of digital content — photos, videos, and user interactions — stored, accessed, and transmitted by Internet services. To better handle such content, popular Web services, such as Facebook, have deployed large cache tiers within their serving stacks to lessen the load on backend systems and to decrease the data-request latency for users.

Designing such cache infrastructures exposes challenges across three dimensions: **(1)** Modern Web workloads differ from earlier traditional workloads, due to the large amount of user-created content, as well as the frequency of updates due to user interactions; therefore, more advanced cache-replacement policies are required to provide sustained high hit-ratios, the key metric for caching performance. **(2)** Flash devices are extensively used due to their cost advantage over DRAM and their significantly higher I/O performance than magnetic disks; however, flash often yields low performance and high wearing costs with the small random writes that advanced caching algorithms tend to generate. **(3)** Load balance is critical for the scalability of an entire cache-server tier; however, different content within the modern Web may have disparate popularities, and the dependent data-partition mechanism is often used to co-locate relative data in favor of advanced application queries. Both scenarios exacerbate the imbalance.

In this dissertation, we use two existing cache systems within the Facebook infrastructure — the photo-cache as a representative of static-content cache, and the TAO cache as an example of in-memory cache solutions — to address the three challenges men-

tioned above.

First, we examine the workload caused by accessing photos on Facebook and the effectiveness of the many layers of photo caches that have been deployed. By analyzing an event stream covering almost 80 million requests for more than 1 million unique photos, we are able to study cache-access patterns, evaluate current cache efficacy, and explore the potential performance benefits of certain advanced eviction algorithms at multiple cache layers.

Second, when building advanced cache on flash devices, in order to address the performance degradation caused by small random writes, we propose the novel RIPQ (Restricted Insertion Priority Queue) framework. RIPQ allows for advanced caching algorithms with large cache sizes, high throughput, and long device lifetime. RIPQ maintains an approximate priority queue efficiently on flash by aggregating small random writes into large writes to a restricted set of insertion points, lazily moving items, and co-locating items with similar priorities. We show that two families of advanced caching algorithms, Segmented-LRU and GDSF (Greedy-Dual-Size-Frequency), can be easily implemented with RIPQ. Our evaluation builds on traces from Facebook's photo-serving stack shows that GDSF algorithms with RIPQ can improve cache hit ratios by ~20% over the current production system, while incurring low overhead and achieving high throughput.

Third, we investigate the principal causes of load imbalance — including data co-location, non-ideal hashing scenario, and hot-spot temporal effects. We analyze Facebook's runtime traffic against the partitioned cache tier in front of TAO, a subsystem that stores objects associated with Facebook's social graph. As part of this investigation, we also employ trace-drive analytics to study the benefits and limitations of current load-balancing methods — including consistent hashing and hot-partition replication (with front-end caching as a special case) — and we suggest areas for future research.

BIOGRAPHICAL SKETCH

Qi Huang grew up in Wuhan, China; he spent the first 25 years of his life there, during which he attended a 4-year undergraduate school, as well as a 3-year taste of graduate school at Huazhong University of Science and Technology (HUST). Qi's initial attraction to a major in Computer Science centered around his rationalization of computer-game time, but later the joy of programming real software for his college mates drove him toward his first graduate-school adventure at Cluster and Grid Computing Lab (CGCL) & Services Computing Technology and System Lab (SCTS). Working with Prof. Hai Jin, Prof. Zhongfen Han, and Prof. Xiaofei Liao, Qi took a ride on the early wave of Peer-to-Peer (P2P) streaming technologies in China, helping to develop a popular streaming service used by thousands of China Education and Research Network (CERNET) users and Internet Protocol Television (IPTV) providers including Samsung, Intel, and UTStarcom.

In 2008, after completing his P2P efforts, Qi took a leave-of-absence from HUST to visit Prof. Ken Birman and Dr. Robbert van Renesse's research group in the Computer Science department of Cornell University. In Spring, 2011, he became a full-time Ph.D. student at Cornell in order to pursue distributed-system studies in the burgeoning sub-field of data-center- and cloud-computing. Since 2012, Qi has collaborated with Facebook to analyze and design distributed caching systems for static content and graph data. That effort has defined the three primary research achievement that comprise the caching story in this dissertation. Qi successfully defended his Ph.D. dissertation in July, 2014.

To my wife Lu Cui, who gives me the courage to chase my dream.

ACKNOWLEDGEMENTS

I am forever indebted to my advisers, Ken Birman and Robbert van Renesse, from whom I learned much about the art and philosophy of systems research, as well as principles for many other aspects of life. Ken pushed me to be an independent researcher from day one, and he always maintained faith in my abilities, even when I doubted myself. His seemingly bottomless well of ideas and energy has made his office my very first stop when I face hurdles. Robbert is an oracle of systems wisdom and a pillar of warm-hearted support. His persistent precision and focus on quality results has forever shaped my work attitude. I leave Cornell with a warm feeling of gratitude for their tutorship.

The Ph.D. journey has been an adventure with many characters and I owe a big debt to my two senior collaborators: Ymir Vigfusson and Daniel A. Freedman. Ymir and Daniel have mentored me on every graduate-school skill hand-in-hand: how to choose research topics, how to plan projects, how to evaluate experimental results, how to structure research papers, how to deliver academic presentations, how to efficiently juggle multiple research demands, and even how to interact with colleagues at conferences. As I reflect back on my graduate years, I can think of many moments where thoughtful feedback and encouraging words from them have helped me steer my course. I would also like to acknowledge Wyatt Lloyd, who was the third senior student mentor I had towards the end of my graduate study. His foundational work on Facebook's photo tracing enabled our later full-stack analysis paper, and his endless assistance ensured that the resulting talk was the most professional and polished presentation I have yet delivered.

Systems research cannot be done in isolation, and I am honored to have had stellar collaborators lending me a helping hand on this dissertation. Linpeng Tang led the cache-on-flash project with amazing diligence and impressive creativity. Helga Gudmundsdottir has played an important role in our load-balance study by implementing the streaming algorithm and running simulations that correspond to very diverse scenar-

ios.

It takes significant effort behind-the-scenes for an industry internship to become a fruitful academic collaboration. I am grateful that I had three great mentors at Facebook who made this happen. All three pieces of work presented in this dissertation truly benefited from their tremendous support: Sanjeev Kumar, Yee Jiun Song, and Philippe Ajoux. I also want to thank the academic duo at Facebook who bridged the connection between Cornell and Facebook: Harry C. Li, and Jessica Taylor.

I come now to my best friend, Lu Cui, who also happens to be my wife. I would never have been able to undertake a Ph.D., or let alone complete it, had it not been for her encouragement and selfless support throughout the entire graduate study period. She made me brave when I felt timid, and confident when I doubted myself; she added her unique love of life to my perspective of the world. With her, my life is enriched every day. Thank you.

I would not have come thus far without the freedom and support given to me by my loving parents, Kanglin Huang and Huilan Tang, who ensured that schooling did not get in the way of my education. I also deeply appreciate my aunt and uncle, Jiulin Huang and Yimei Wu, who raised me for six years as their own child when I had to be away from home for school. Last, but certainly not least, I extend my gratitude to my friends in Ithaca who made my stay at Cornell in every aspect more welcoming, exciting, and enjoyable. I am honored to have had your company.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Since the late 90s, the interface of World Wide Web has shifted increasingly from static hypertext pages to a dynamic content exchange platform. The switch favors modern applications, such as social networking, wikipedia, and video sharing sites. In these new generation of Web portals, users are not limited to the passive viewing of pre-generated content, instead they are able to interact and collaborate with one another. As a consequence the rise of modern Web applications has also seen a surge in the amount of digital content — static content including photos, and videos, as well as dynamic content such as user interactions — stored and served by the Web. To better serve this content, modern Web providers have designed and deployed large caching tiers within their serving stack (including static-content cache for media binaries, and in-memory cache for dynamic user interactions) to lessen the load on their backend systems and to decrease the data request latency for users. As the caching infrastructure becomes critical to scale a modern Web application, this dissertation seeks to analyze current caching performance and to explore solutions for subsequent caching designs. In this study, we analyze the runtime behavior of a static-content caching stack, and an in-memory caching solution; using existing systems that serve production traffic for a large modern Web site. Through analysis, we reveal valuable insights on the modern Web caching workload, evaluate their performance efficacy, and propose novel designs to build advanced caching solution for both static-content cache and in-memory cache.

1.1 Static-Content Caching Analysis and Design

Among the many forms of digital contents that users upload and access on modern Web applications, the media binary large objects (BLOBs) are the most prevalent in terms

of storage footprint and bandwidth consumption on the Internet. As a result, the effectiveness of the stacks that store and deliver static-content has become an important issue for the Web provider community [16, 17]. Although there has been a large collection of studies on content-serving in other settings [15, 25, 36, 39, 41, 81, 70, 77, 84, 86], workload within the new generation Web applications remains mysterious for much of the academic community, and the lack of insights prevents further studies on system designs.

To better understand the impact of modern Web application workloads on the static-content serving system, especially the caching tier, this dissertation first explores the dynamics of a full content-serving stack in production, Facebook's photo-serving stack. Facebook's photo-serving stack is complex and geographically distributed. It includes; browser caches for every client, Edge Caches at ~20 PoPs, an Origin Cache, and an underlying storage layer that is widely distributed across multiple data centers. We modified every Facebook-controlled layer of the stack and sampled the resulting event stream to obtain traces covering over 77 million requests for more than 1 million unique photos. This permits us to study the cache access patterns, geolocation of clients and servers; explore the potential performance benefits of coordinating Edge caches as well as adopting advanced eviction algorithms at both Edge and Origin layers.

Our results (1) quantify the overall traffic percentages served by different layers: 65.5% browser cache, 20.0% Edge Cache, 4.6% Origin Cache, and 9.9% Backend storage, (2) reveal that a significant portion of photo requests are routed to remote PoPs and data centers as a consequence both of load-balancing and peering policy, (3) show that the popularity of photos is highly dependent on content age and conditionally dependent on the social-networking metrics we considered, and (4) demonstrate the potential performance benefits of coordinating Edge Caches and adopting advanced eviction al-

gorithms at both Edge and Origin layers. The detailed analysis study is presented further in Chapter 3.

While adopting advanced caching eviction algorithms can directly improve the caching performance, the underlying caching media — flash device — makes this challenging because advanced caching algorithms generate many small random writes. Even though all photo requests from users are reads, each miss in the caching tier triggers a request to the backend to generate the requested content, which in turn causes a write operation. With moderate hit ratios reported by Facebook's photo cache, the workload on cache can be considered write-heavy. Unfortunately, the Flash Translation Layer (FTL) on modern flash device performs poorly with such workloads, resulting in lower throughput and decreased device lifespan. For these reasons, the production system at Facebook employs a First-In-First-Out (FIFO) caching algorithm, for which the replacement policy generates sequential writes.

To resolve this challenge, this dissertation presents a flash cache design, named Restricted Insertion Priority Queue (RIPQ), which allows advanced caching algorithms without paying the device penalty. RIPQ provides an approximate priority queue interface; a convenient abstraction for implementing several advanced caching algorithms [22, 87]. Through key mechanisms of limiting insertion points, lazily moving items, and co-locating items with similar priorities, RIPQ is able to support the priority queue interface with mostly consolidated large writes to flash.

With RIPQ support, the evaluation of Facebook's photo workloads shows that both Segmented Least-Recently-Used (LRU) algorithm [54] and Greedy-Dual-Size-Frequency (GDSF) algorithm implementations achieve significantly higher hit ratios than the FIFO cache at Facebook. Specifically, GDFS algorithms with RIPQ improves the hit ratio by 16.7-20% at Origin cache. Such improvements translate to 22.5-30%

I/O-Operations-Per-Second (IOPS) reduction to the backend storage tier. RIPQ also efficiently and faithfully implements these two algorithms on flash with 90% utilization of the device, incurs 1.2X write amplification, and achieves over 12K req/sec throughput. The detailed study of RIPQ is further presented in Chapter 4.

1.2 In-Memory Caching Analysis and Design

Caches for dynamic content are also important for the scalability of modern Web services because they can decrease request latency for users and relieve load on storage and database servers. There is a key distinction between the data that is cached in these scenarios: static-content caches mainly store larger BLOBs with simple read and write operations, whereas caches for dynamic content are built for shorter structural data that is derived from database queries and application transformations. Therefore while static-content caches can generally afford to look up data on slower secondary storage like a magnetic disk or a flash device, caches for dynamic content require low latency and are thus generally stored in DRAM. We will call the latter in-memory caches for short.

In addition, the complexity of in-memory caches has grown in tandem with their popularity among various storage solutions. Today's high-performance storage systems have borrowed a rich set of features from traditional databases, including: transactions, consistency guarantees, richer data types [33], as well as operations such as joins [57] and range queries [20]. In order to support such diverse features, the design of in-memory cache has also become increasingly complicated.

In this dissertation, we focus on one common issue, load imbalance, which is shared among in-memory caching systems with advanced operation support. To prevent the ag-

gregate request volume at a modern Web site from overwhelming a single cache server, cached data is normally partitioned among hundreds or thousands of cache servers. However, partitions confronted with real-world workloads often sustain variable and dynamic request rates that can contribute to significant load imbalance.

Although there are many options for general load balancing, to the best of our knowledge no comprehensive study to date permits online analysis of the key culprits based on a real workload, or provides justifications on the limit of each approach. To fill this knowledge gap, we characterize the load imbalance status at Facebook's social-graph cache, TAO, where data have skewed access popularity and cannot randomly be separated due to the range query interface. We also explore how different categories of approaches — fine-tuned consistent hashing and hot content replication (including a special case for front-end cache) — might help to mitigate their impact and investigate the limitation of each approach category. Chapter 5 will further explore the detailed load imbalance study on in-memory cache.

1.3 Contribution

In this dissertation we analyze two important types of caching infrastructures — a static-content cache for media binaries, and an in-memory cache for dynamic user interactions — that are critical to support and scale modern Web applications. We then propose novel design solutions for each.

For static-content caching solutions, our study addresses two issues: (1) a lack of understanding of workload patterns within the new generation Web, and (2) a caching design that takes full benefit of modern hardware (such as flash devices), without losing the flexibility to be optimized for certain workloads. In this study, we use Facebook's

photo-serving stack as a real-world study case. However, our contributions also apply to a broader domain of modern Web that serves static content, including photo and video.

For in-memory caching solutions, our study focuses on the issue of load imbalance, which is widely reported as a critical hurdle to scale Web traffic in terms of number of requests. In this work, we use Facebook's TAO, a cached storage that stores user interactions on the social network graph, as another real-world study case. Results from this study also apply to other in-memory caching deployments that store data of disparate popularity and co-locate dependent data favored by advanced features including range queries.

Our contributions include:

- We conduct a full-stack analysis on a modern Website's photo-serving stack. To the best of our knowledge, this analysis is the first study to examine an entire Internet image-serving infrastructure at a massive scale.
  - Through the analysis, we are able to quantify a layer-by-layer traffic pattern towards a large collection of photos and its performance impact on different caching tiers.
  - Moreover, we determine that the caching performance in terms of hit ratio can be significantly improved by collaborating geographic-scale caches and adopting advanced caching algorithms such as Segmented LRU.
  - By examining the relationship between content access and associated metadata in the social network application, we also find that content popularity is strongly correlated with age and is conditionally dependent on the owner's social connectivity. It points to an interesting direction for building even better caching policies with the help of application knowledge.

- We propose a novel design on Restricted-Insertion-Priority-Queue (RIPQ), an approximate priority queue that can support advanced caching algorithms efficiently on modern flash devices.
  - We demonstrate the flexibility of approximate priority queue interface by implementing two advanced caching algorithm families — Segmented LRU, and GDSF — on top of RIPQ.
  - We evaluate RIPQ-based caching performance gain on the FusionIO device with the photo-serving trace we collected from the previous analysis. With RIPQ, advanced caching algorithms can be implemented on modern flash devices with high algorithm fidelity, high space utilization, while still achieving full write performance and low device overhead.
  - To overcome the memory constraint of RIPQ, we also provide a simplified Single-Insertion-Priority-Queue (SIPQ) design, which consumes much less memory than RIPQ, and is capable to support simple caching algorithms like LRU.
  - The design of RIPQ, as well as its simplified version with single insertion (SIPQ), are both novel in the unexplored space of flash-based advanced caching for static-content.
- We also investigate load imbalance within a cluster of in-memory caching servers, where data have skewed access popularity and cannot be randomly separated.
  - We determine that the existing load imbalance among in-memory caching servers can stem from a combination of load-insensitive partitioning, extremely hot shards, and random temporal effects.
  - In contrast, the popularity skewness among different objects is not a major cause of cache load imbalance.
  - We conduct a survey of current approaches to load balancing with real-world

traces simulation. Our results would help guide the direction of future research regarding in-memory caching load balancing.

This dissertation continues with background and related work in Chapter 2. We demonstrate the analysis study on Facebook's photo-serving stack, which is a large-scale static-content serving system with multiple caching tiers, in Chapter 3. We then present the design of RIPQ, an advanced static-content caching framework for modern flash devices, in Chapter 4. The investigation effort on characterizing in-memory cache load-imbalance is included in Chapter 5. We discuss future work on open questions and conclude in Chapter 6.

CHAPTER 2

BACKGROUND & RELATED WORK

2.1 Modern Web Architecture

Starting from the late 90s, the interface of World Wide Web has evolved from static hypertext pages to a dynamic content exchange platform. The switch favors modern applications, such as social networking, wikipedia, and video sharing sites. In these new generation Web portals, users are not limited to the passive viewing of pre-generated content, instead they are able to interact and collaborate with one another. As a consequence the complexity of Web architecture design has grown together with the demand of infrastructure support for today's modern Web applications. Figure 2.1 demonstrates a typical modern Web application's architecture and its work flow.

Web Front-End. When a client — browser, or mobile application — accesses a Web service, it sends out a HTTP request and the first tier of servers on the receiving side is Web front-end. A Web server equipped with Apache [2], Nginx [6], or other customized Web language execution engine such as HHVM [3], is responsible for serving this request. It decodes the request information, triggers appropriate application logic to fetch content updates, and then renders a Web response to send back to the client. A modern website often tailors its content experience for each specific user, and it requires additional platform information provided by the client, such as the device location and screen size. Once the client embeds this information in the user request, Web front-end is responsible for extracting it for the later application logic. Moreover, modern websites are often composed of multiple application modules to provide a set of features useful for its clients. For instance, Facebook's homepage contains Newsfeed, People

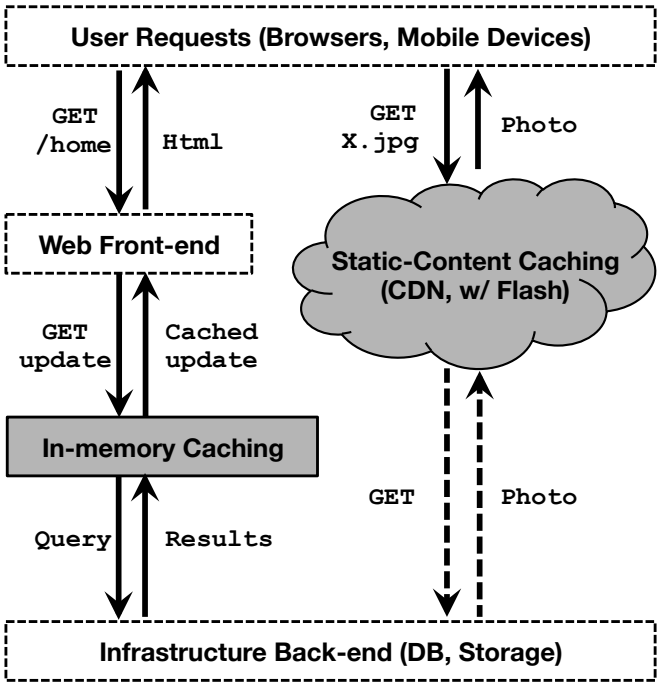


Figure 2.1: Modern Web architecture. When the Web front-end receives a user request, it starts to generate a response to include the most recent updates for that user. The in-memory cache keeps the most popular update query results so that the fetching operation does not always hit the back-end. For static content, Web front-ends would redirect the client to go through a dedicated stack that is often composed of flash-equipped caching layers.

You May Know, Messenger, and Notification; each operates separately based on the different social-network activities from the same user. While some applications are simple enough to be purely executed on the Web front-end itself, others depend on further content that Web servers need to fetch from in-memory caches or even back-end storage.

Infrastructure Back-End. Some complicated application modules may have their own back-ends, each as an additional mini website built to support a single function. However, common needs for the entire Web application are better addressed by a shared infrastructure, which can be further optimized at scale. A typical infrastructure back-end is the distributed BLOB storage, which stores non-structural media content such as photos and videos. Although different application modules may process this content in

various manners, the access interface towards the raw data remains similar. It allows all BLOBs to be maintained by a single system that only focuses on efficient data fetching for the common interface. With an efficient infrastructure back-end in place, application back-ends are free from common concerns, such as data loss for media content, and can be simplified to cater the sole need from a single application.

In-Memory Cache. In order to reduce data fetching latency and back-end database querying overhead, Web servers' requests are first directed to a tier of caches, where popular content results reside in DRAM. Only if the requested data does not reside in the caching space would a data request reach the back-end. Based on the manner in which the data is queried, there are two categories of caching interfaces: key-value and advanced query. With key-value interface, query results from the back-end are cached as independent objects and indexed with hashed keys. Only a request with the exact key can be satisfied with a hit. To support advanced queries, the data must retain richer semantic structure than simply a key and a value. However, the additional metadata could allow data in the cache to be subjected to different queries and thereby avoid additional cache misses. `memcached` [5] and TAO [20] (the graph storage solution at Facebook) are examples of these two styles, respectively.

Static-Content Cache. The popularity of modern Web services has driven a dramatic surge in the amount of user-created content, especially static media binaries, stored by Web portals. As a result, the effectiveness of the Web architecture that delivers static content has become an important issue for the Web service provider. In contrast to dynamic Web response that has to be generated on the fly by Web servers, static content rarely changes and as a result can be served further away from the Web server as long as a copy of the data is in place. To reduce the user-perceived latency for downloading

static content, the Content Delivery Network (CDN) is often deployed as a static-content caching solution, which stores popular content at geographically distributed caching facilities that are close to users. When a client requests a photo, or a video, it first tries to fetch the data from CDN, which in turn fetches the data from the infrastructure back-end when the content is absent. Comparing to the structural data, media binaries have larger storage footprint, thus static-content caching solutions use flash devices extensively because of its cost advantages over DRAM and higher I/O performance than magnetic disks.

In particular, this dissertation focus on the analysis and design of advanced caching solutions, which include both the static-content caching and the in-memory caching components within the modern Web architecture. These two components have also been highlighted with grey in Figure 2.1.

2.2 Static-Content Cache

A static-content serving stack often deploys multiple caching layers that are organized as a CDN) to reduce back-end load and content delivery latency. Figure 2.2 shows a typical, simplified stack that contains three layers of caches: Edge, Midgress, and Origin. At each cache site, individual cache objects are hashed to different caching machines according to their URI. Each caching machine then functions as an independent cache for its subset of objects.

Edge Cache. The *Edge cache layer* operates at the edge of the Internet. It consists of caching facilities that are deployed at geographically distributed Internet Points of Presences (PoPs), and serves as the front caching tier to which clients' requests are

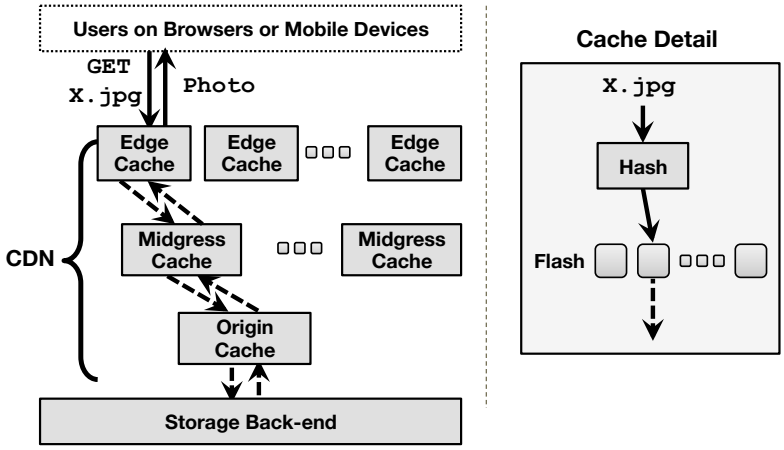


Figure 2.2: A typical static-content serving stack. Requests are directed through a CDN with layers of caches. Each cache hashes objects to a bucket associated with a flash equipped server.

routed. Although edge caches from different locations can collaborate with one another, a typical deployment operates the cache site within each PoP as an independent caching component. Within each PoP, individual cache objects are hashed to different caching machines according to their Uniform-Resource-Identifier (URI), and each caching machine functions for its subset of objects. The main objective of each Edge cache site is to place the popular content at a place, such as PoP, that is close to the end user in terms of the routing distance on the Internet. It helps to reduce the latency of delivering such content from a further location, as well as the bandwidth consumption between the user-end network and Web application provider's network. As a result, the major metric for the edge cache performance is its byte-wise hit ratio: the bandwidth consumed by serving content in cache versus the bandwidth consumed by serving all requests to the cache. When the edge cache receives a request for a content not in cache, it serves as a proxy to fetch the data from downstream.

Midgress Cache. When the edge cache deployment grows beyond the continent limit, some sites become too remote that every miss at the edge cache takes an unacceptable

amount of time. Deploying an entire data center with full stack of Web front-ends and infrastructure back-ends to backup these Edges can be unnecessarily expensive; thus a more appealing alternative is to deploy a *Midgress cache layer*. The midgress cache is very similar to the edge cache site in terms of function and performance metric, and sometimes a large edge cache can be used as a midgress for other smaller edge sites nearby. In contrast to edge caches, the midgress layer is often optional.

Origin Cache. The *Origin cache layer* serves as the last resort to protect the disk-based storage back-end. As this layer is often co-located with the storage system, origin cache servers are built to shelter the excessive I/O load that magnetic disks could be throttled by. As a result, the major performance metric for the origin is object-wise hit ratio: the number of requests served by the content in cache versus the total number of requests sent to the cache. Even when machines are distributed at geographically distant locations, the orign cache layer is operated as a single cache component in order to maximize the object-wise hit ratio.

Facing high request rates for a large set of objects, all three caching layers are often equipped with enterprise-level flash drives. Flash offers much higher capacity than DRAM and far higher IOPS performance than magnetic disks

2.3 In-Memory Cache

When a front-end Web server receives Web clients' HTTP requests, it often spawns numerous data fetching requests to generate a content-rich response. To reduce the latency of data fetching and database querying overhead, such data fetches are first directed to a tier of caches, where popular content results reside in DRAM. Only if the requested

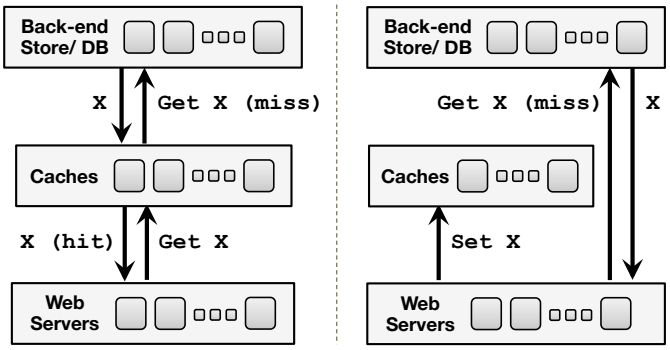


Figure 2.3: Cache architecture. Left shows how read-through cache operates between the front-end Web servers and the back-end; right shows how miss is handled for the read-aside cache.

data does not reside in the caching space would a data request reach the back-end. Based on the manner in which the back-end request is redirected, there are two categories of caching tiers: read-through and read-aside. Figure 2.3 demonstrates the request flow of both styles: the left plot follows the read-through style, within which the cache serves as the proxy to fetch data from the back-end while caching is on the way; the right plot follows the read-aside style, within which the Web server takes the responsibility to request the data from back-end servers and fills the content in cache later. TAO (the graph storage solution at Facebook) organizes its cache tier as read-through caches, and specifically has two layers of caches: follower and leader. The follower cluster co-locates with each Web front-end cluster, while a leader tier cluster serves an entire region's followers. `memcached` [5] is a popular caching solution which follows the read-aside style.

2.4 Related Work

2.4.1 Static-Content Stack Analysis

Many measurement studies have examined web access patterns for services associated with content delivery, storage, and web hosting. To the best of our knowledge, our study is the first to systematically instrument and analyze a real-world workload at the scale as large as that of Facebook, and to successfully trace such a high volume of events throughout a massively distributed stack. The most closely related prior work that we identified is a classic study by Saroiu et al. [77] that compared the characteristics of four different Internet content delivery mechanisms using a network trace captured between University of Washington and the rest of the Internet. That work was undertaken some time ago, however, during a period when peer-to-peer networks were a dominant source of Internet traffic. A follow-on paper [39] took the analysis further, comparing the media popularity distribution seen in the campus trace with that associated with traditional web traffic. Our focus on BLOB traffic induced by social networking thus looks at a different question, and on a much larger scale.

Additional work involved studies of the *flash crowd phenomenon* in content delivery networks (CDNs) [86, 52, 78]. These efforts focused on data obtained by monitoring aggregated network traffic. Such work yields broad statistics, but we gain only limited insight into application properties that gave rise to the phenomena observed. An exception is Freedman's investigation [36] of a 5-year system log of the Coral CDN [37], studying its behavior with detailed insight into its operational properties and architecture. Our work covers both sides, enabling us to break behaviors down in a manner not previously possible.

Whereas our focus here was on the network side of the image-processing stack, the lowest layer we considered is the back-end storage server. Here, one can point to many classic studies [15, 25, 41, 70, 84] and also to the recent detailed architecture and performance evaluation of Haystack, the Facebook BLOB storage server [16]. Detailed network and caching traffic traces can inform the design of future storage systems, just as they enabled us to study how different caching policies might reduce loads within the Facebook infrastructure. However, constraints of length and focus forced us to limit the scope of the present study, and we leave this for future investigation.

Numerous research projects have explored the modelling of web workload, and several recent papers [47, 21] monitor web traffic over extended time periods, to the extent of evaluating workload changes as the web itself evolved. Breslau et al. [19] explores the impact of Zipf's law with respect to web caching, showing that Zipf-like popularity distributions cause cache hit rates to grow logarithmically with population size, as well as other effects. In contrast, Guo et al. [40] argues that access to media content often has a significantly distorted head and tail relative to a classic Zipf distribution. We found that caches closest to the client browser have a purely Zipf popularity distribution, but that deep within the Facebook architecture, the Haystack back-end experiences a workload that is similar to that which characterizes a stretched exponential distribution [40].

2.4.2 Flash-aware Cache and Storage

To the best of our knowledge, there is no existing work that provides a flexible framework for efficiently implementing advanced static-content caching on flash. However, RIPQ sits at the intersection of several heavily-researched fields: RAM-based caching, Flash-based storage, Flash performance studies, and priority queues.

RAM-based Caching Caching has been an important research topic since the early days of computer science and many algorithms have been proposed to better capture the characteristics of different workloads. Some well-known features include recency (LRU, MRU) [27], frequency (LFU) [65], inter-reference time (LIRS) [49], and size (SIZE) [8]. There have also been a plethora of more advanced algorithms that consider multiple features, such as Multi-Queue [89] and Segmented LRU [54] for both recency and frequency, Greedy-Dual [88] and its variants like Greedy-Dual-Size [22] (GDS) and Greedy-Dual-Size-Frequency [26] (GDSF) for a more general method to compose the expected miss cost and minimize it.

While more advanced algorithms can potentially yield significant performance improvements, such as Segmented-LRU and GDSF for Facebook photo workload, a gap still remains for efficient implementations on top of flash devices because most algorithms are hardware-agnostic: they implicitly assume data can be moved and overwritten with little overhead. Such assumptions do not hold on modern flash due to its asymmetric costs for reads and writes and the performance deterioration caused by its internal garbage collection.

Recently more flash-aware caching algorithms have also been proposed [74, 58, 59, 83, 45]. While most of them still treat flash as the back-end storage of a RAM cache and focus on shaping the cache-storage workload to be more flash-friendly, a few can only implement a single algorithm on flash with no flexibility. Other recent work has used flash as a write-back block cache [60]. Our work on RIPQ and SIPQ provides an efficient way to directly manage the entire flash capacity as the caching space, which is crucial for scenarios such as static-content delivery systems with large working-sets, and also provides flexibility in the choice of caching algorithm.

Flash-based Store Many flash-based storage systems, especially key-value stores have been recently proposed to work efficiently on flash hardware. Systems such as FAWN-KV [12], SILT [64], LevelDB [4], and RocksDB [7] group write operations from an upper layer and only flush to the device using sequential writes. However, being designed to cater to read-heavy workloads and other performance/application metrics such as memory footprints and range-query efficiencies, these systems make trade-offs (such as conducting on-flash data sorting and merges), that would yield high device overhead for write-heavy workloads. In contrast, RIPQ and SIPQ are specifically optimized for a (random) write-heavy workload and only support caching-required interfaces. Our narrowly scoped scenario gives RIPQ and SIPQ great flexibility to avoid device overhead as demonstrated in the dissertation.

Because caching algorithms can be built on top of more general flash-based storage systems, we chose RocksDB [7] as a sample baseline in evaluation to demonstrate its disadvantages for our scenario explicitly: (1) items are not grouped by their caching priorities like RIPQ, and (2) the key-value store interface does not provide the flexibility to access items that are marked removable but not removed yet as in SIPQ. These two disadvantages result in a worse trade-off space between caching performance and write-amplification compared to the RIPQ and SIPQ frameworks.

Study on Flash Performance and Interface While flash hardware itself is also an important topic, studies that examine the application perceived performance and interface are more related to our work. For instance, previous research [53, 18, 80, 67] that reports the random write performance deterioration on flash helps verify our observations in the flash performance study.

Systematic approaches to mitigate this specific problem have also been previously

proposed at different levels, such as separating the treatment of cold and hot data in the FTL by LAST [62], and the similar technique in filesystem by SFS [67]. These approaches work well for skewed write workloads where only a small subset of the data is hot and updated often, and thus can be grouped together for garbage collection with lower overhead. In RIPQ, cached contents are explicitly tagged with priority values that indicate their hotness, and are co-located within the same device block if their priority values are close. In a sense, such priorities provide a *prior* for identifying content hotness.

Recently, a more holistic approach known as Software-Defined Flash [71] (SDF) has also been proposed to improve the performance of flash hardware for specialized workloads. SDF takes a step back from the FTL and exposes the internal flash structure to make explicit data arrangement controllable by the application. Leveraging SDF-like interfaces would enable RIPQ to further reduce its memory consumption.

Priority Queue Both RIPQ and SIPQ rely on the priority queue abstract data type and the design of priority queues with different performance characteristics have been a classic topic in theoretical computer science as well [10, 35, 24]. Instead of building an exact priority queue, RIPQ uses an approximation to trade algorithm fidelity for flash-aware optimization.

2.4.3 Load Balance of Distributed Cache and Storage

As modern Web application becomes more data intensive, its infrastructure back-end — such as distributed cache and storage — also grows in scale. These back-end systems scale using a combination of partitioning (spreading data or requests across multiple

servers, where each server handles a subset of the overall load requirement) and replication. In the case of Web application logic that relies on the data from multiple servers, the data-fetching overhead, in terms of time, is equal to the data-response time of the slowest server. And as a result, the capacity threshold of an entire caching or storage tier to meet its latency and throughput goals is often determined by the busiest server. For this reason, a good load balance is necessary to ensure that no server's resource is underutilized when the entire tier is considered overloaded.

There are two components of the load within a distributed cache or storage system: the storage capacity measured in terms of the number of objects, and the traffic load measured in terms of the number of requests. Ideally, data should be spread uniformly among servers, and no server should be responsible for more requests than another server.

Capacity is typically load-balanced by striping the data [75], or by hashing the ID space of all stored objects. The latter option, especially consistent hashing [55] is popular due to its simplicity and stability in the presence of server dynamics. Consistent hashing is often used together with virtual nodes, where each physical server acts as several different servers in the consistent hashing ring [30] to improve the quality of capacity load balance.

Request load is typically load-balanced in one of three ways. First, some systems dynamically shift data from busy servers to less busy servers to balance the request load [79]. Second, front-end caching is also proved to be efficient in smoothening a popularity skewed workload towards a distributed storage system behind the cache [34]. Systems using the third method, such as Mitzenmacher's well-known "power of two choices" load balancing, rely upon replication to be able to direct data requests to the least-loaded of two or more replicas, substantially improving load balance in the pro-

cess [68]. A recent study tackling the load-imbalance in `memcached` applies the replication technique [42].

Some large scale systems have also applied combinations of these mechanisms, e.g., Facebook's TAO [20] uses front-end caching, consistent hashing, virtual nodes, and replication. However, our study finds that the busiest server has more than 2 times the load of the least busy server within the same cluster. Load imbalance remains an important challenge for partitioned services [31].

CHAPTER 3

ANALYSIS OF FACEBOOK'S PHOTO CACHING

3.1 Introduction

In this chapter, we present the detailed analysis on Facebook's photo-serving stack, from the client browser to Facebook's Haystack storage server, looking both at the performance of each layer and at interactions between multiple system layers. The goal of this study is to gain insights that can inform design decisions for future content caching, storage, and delivery systems. Specifically, we ask (1) how much of the access traffic is ultimately served by the Backend storage server, as opposed to the many caching layers between the browser and the Backend, (2) how requests travel through the overall photo-serving stack, (3) how different cache sizes and eviction algorithms would affect the current performance, and (4) what object meta data is most predictive of subsequent access patterns.

Our study addresses these questions by collecting and correlating access records from multiple layers of the Facebook Internet hierarchy between clients and Backend storage servers. The instrumented components include client browsers running on all desktops and laptops accessing the social network website, all Edge cache hosts deployed at geographically distributed points of presence (PoP), the Origin cache in US data centers, and Backend servers residing in US data centers. This enabled us to study the traffic distribution at each layer, and the relationship between the events observed and such factors as cache effects, geographical location (for client, Edge PoP and data center) and content properties such as content age and the owner's social connectivity. This data set also enables us to simulate caching performance with various cache sizes and eviction algorithms. We focus on what we identified as key questions in shaping a

new generation of static-content serving infrastructure solutions.

1. To the best of our knowledge, our analysis is the first study to examine an entire Internet image-serving infrastructure at a massive scale.
2. By quantifying layer-by-layer cache effectiveness, we find that browser caches, Edge caches and the Origin cache handle an aggregated 90% of the traffic. For the most-popular 0.03% of content, cache hit rates neared 100%. This narrow but high success rate reshapes the load patterns for Backend servers, which see approximately Zipfian traffic but with Zipf coefficient α diminishing deeper in the stack.
3. By looking at geographical traffic flow from clients to Backend, we find that content is often served across a large distance rather than locally.
4. We identify opportunities to improve cache hit ratios using geographic-scale collaborative caching at Edge servers, and by adopting advanced eviction algorithms such as S4LRU in the Edge and Origin.
5. By examining the relationship between image access and associated meta-data, we find that content popularity rapidly drops with age following a Pareto distribution and is conditionally dependent on the owner's social connectivity.

The chapter is organized as follows. Section 3.2 presents an overview of the Facebook photo serving-stack, highlighting our instrumentation points. Section 3.3 describes our sampling methodology. After giving a high level overview of the workload characteristics and current caching performance in Section 3.4, Sections 3.5, 3.7, and 3.6 further break down the analysis in three categories: geographical traffic distribution, traffic association with content age and the content owners' social connectivity, and potential improvements.

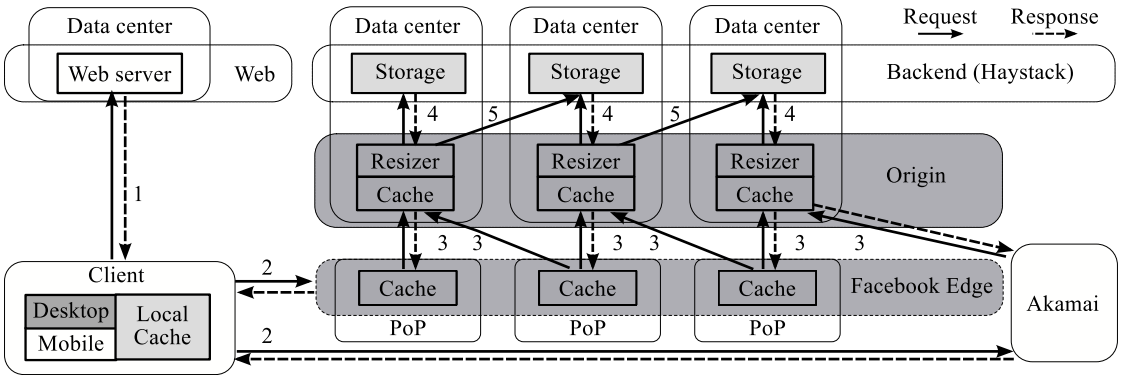


Figure 3.1: Facebook photo serving stack: components are linked to show the photo retrieval work-flow. *Desktop* and *Mobile* clients initiate request traffic, which routes either directly to the Facebook Edge or via Akamai depending on the fetch path. The Origin Cache collects traffic from both paths, serving images from its cache and resizing them if needed. The Haystack backend holds the actual image content. Shading highlights components tracked directly (dark) or indirectly (light) in our measurement infrastructure.

3.2 Facebook's Photo-Serving Stack

As today's largest social-networking provider, Facebook stores and serves billions of photos on behalf of users. To deliver this content efficiently, with high availability and low latency, Facebook operates a massive photo-serving stack distributed at geographic scale. The sheer size of the resulting infrastructure and the high loads it continuously serves make it challenging to instrument. At a typical moment in time there may be hundreds of millions of clients interacting with Facebook Edge Caches. These are backed by Origin Cache and Haystack storage systems running in data centers located worldwide. To maximize availability and give Facebook's routing infrastructure as much freedom as possible, all of these components are capable of responding to any photo-access request. This architecture and the full life cycle of a photo request are shown in Figure 3.1; shaded elements designate the components accessible in this study.

3.2.1 The Facebook Photo-Caching Stack

When a user receives an HTML file from Facebook's front-end web servers (step 1), a browser or mobile client app begins downloading photos based on the embedded URLs in that file. These URLs are custom-generated by web servers to control traffic distribution across the serving stack: they include a unique photo identifier, specify the display dimensions of the image, and encode the *fetch path*, which specifies where a request that misses at each layer of cache should be directed next. Once there is a hit at any layer, the photo is sent back in reverse along the fetch path and then returned to the client.

There are two parallel stacks that cache photos, one run by Akamai and one by Facebook. For this study, we focus on accesses originating at locations for which Facebook's infrastructure serves all requests, ensuring that the data reported here has no bias associated with our lack of instrumentation for the Akamai stack. The remainder of this section describes Facebook's stack.

There are three layers of caches in front of the backend servers that store the actual photos. These caches, ordered by their proximity to clients, are the client browser's cache, an Edge Cache, and the Origin Cache.

Browser The first cache layer is in the client's browser. The typical browser cache is co-located with the client, uses an in-memory hash table to test for existence in the cache, stores objects on disk, and uses the LRU eviction algorithm. There are, however, many variations on the typical browser cache. If a request misses at the browser cache, the browser sends an HTTP request out to the Internet (step 2). The fetch path dictates whether that request is sent to the Akamai CDN or the Facebook Edge.

Edge The Facebook Edge is comprised of a set of Edge Caches that each run inside points of presence (PoPs) close to end users. There are a small number of Edge Caches spread across the US that all function independently. (As of this study there are nine high-volume Edge Caches, though this number is growing and they are being expanded internationally.) The particular Edge Cache that a request encounters is determined by its fetch path. Each Edge Cache has an in-memory hash table that holds metadata about stored photos and large amounts of flash memory that store the actual photos [38]. If a request hits, it is retrieved from the flash and returned to the client browser. If it misses, the photo is fetched from Facebook's Origin Cache (step 3) and inserted into this Edge Cache. The Edge caches currently all use a FIFO cache replacement policy.

Origin Requests are routed from Edge Caches to servers in the Origin Cache using a hash mapping based on the unique id of the photo being accessed. Like the Edge Caches, each Origin Cache server has an in-memory hash table that holds metadata about stored photos and a large flash memory that stores the actual photos. It uses a FIFO eviction policy.

Haystack The backend, or Haystack, layer is accessed when there is a miss in the Origin cache. Because Origin servers are co-located with storage servers, the image can often be retrieved from a local Haystack server (step 4). If the local copy is held by an overloaded storage server or is unavailable due to system failures, maintenance, or some other issue, the Origin will instead fetch the information from a local replica if one is available. Should there be no locally available replica, the Origin redirects the request to a remote data center.

Haystack resides at the lowest level of the photo serving stack and uses a compact blob representation, storing images within larger segments that are kept on log-

structured volumes. The architecture is optimized to minimize I/O: the system keeps photo volume ids and offsets in memory, performing a single seek and a single disk read to retrieve desired data [16].

3.2.2 Photo Transformations

Facebook serves photos in many different forms to many different users. For instance, a desktop user with a big window will see larger photos than a desktop users with a smaller window who in turn sees larger photos than a mobile user. The resizing and cropping of photos complicates the simple picture of the caching stack we have painted thus far.

In the current architecture all transformations are done between the backend and caching layers, and thus all transformations of an image are treated as independent blobs. As a result, a single cache may have many transformation of the same photo. These transformations are done by Resizers (shown closest to the backend server in Figure 3.1), which are co-located with Origin Cache servers. The Resizers also transform photos that are requested by the Akamai CDN, though the results of those transformations are not stored in the Origin Cache.

When photos are first uploaded to Facebook they are scaled to a small number of common, known sizes, and copies at each of these sizes are saved to the backend Haystack machines. Requests for photos include not only the exact size and cropping requested, but also the original size from which it should be derived. The caching infrastructure treats all of these transformed and cropped photos as separate objects. One opportunity created by our instrumentation is that it lets us explore hypothetical alternatives to this architecture. For example, we evaluated the impact of a redesign that pushes

all resizing actions to the client systems in Section 3.7.

3.2.3 Objective of the Caching Stack

The goals of the Facebook photo-caching stack differ by layer. The primary goal of the Edge cache is to reduce bandwidth between the Edge and Origin datacenters, whereas the main goal for other caches is *traffic sheltering* for its backend Haystack servers, which are I/O bound. This prioritization drives a number of decisions throughout the stack. For example, Facebook opted to treat the Origin cache as a single entity spread across multiple data centers. Doing so maximizes hit rate, and thus the degree of traffic sheltering, even though the design sometimes requires Edge Caches on the East Coast to request data from Origin Cache servers on the West Coast, which increases latency.

3.3 Methodology

We instrumented Facebook's photo-serving infrastructure, gathered a month-long trace, and then analyzed that trace using batch processing. This section presents our data gathering process, explains our sampling methodology, and addresses privacy considerations.

3.3.1 Multi-Point Data Collection

In order to track events through all the layers of the Facebook stack it is necessary to start by independently instrumenting the various components of the stack, collecting a representative sample in a manner that permits correlation of events related to the

same request even when they occur at widely distributed locations in the hierarchy (Figure 3.1). The ability to correlate events across different layers provides new types of insights:

- **Traffic sheltering:** We are able to quantify the degree to which each layer of cache shelters the systems downstream from it. Our data set enables us to distinguish hits, misses, and the corresponding network traffic from the browser caches resident with millions of users down through the Edge Caches, the Origin Cache, and finally to the Backend servers. This type of analysis would not be possible with instrumentation solely at the browser or on the Facebook Edge.
- **Geographical flow:** We can map the geographical flow of requests as they are routed from clients to the layer that resolves them. In some cases requests follow surprisingly remote routes: for example, we found that a significant percentage of requests are routed across the US. Our methodology enables us to evaluate the effectiveness of geoscale load balancing and of the caching hierarchy in light of the observed pattern of traffic.

Client To track requests with minimal code changes, we limit our instrumentation to desktop clients and exclude mobile platforms: (1) all browsers use the same web code base, hence there is no need to write separate code for different platforms; and (2) after a code rollout through Facebook's web servers, all desktop users will start running that new code; an app update takes effect far more slowly. Our client-side component is a fragment of javascript that records when browsers load specific photos that are selected based on a tunable sampling rate. Periodically, the javascript uploads its records to a remote web server and then deletes them locally.

The web servers aggregate results from multiple clients before reporting them to

Scribe [51], a distributed logging service. Because our instrumentation has no visibility into the Akamai infrastructure, we limit data collection to requests for which Facebook serves all traffic; selected to generate a fully representative workload. By correlating the client logs with the logs collected on the Edge cache, we can now trace requests through the entire system.

Edge Cache Much like the client systems, each Edge host reports sampled events to Scribe whenever an HTTP response is sent back to the client. This allows us to learn whether the associated request is a hit or a miss on the Edge, along with other details. When a miss happens, the downstream protocol requires that the hit/miss status at Origin servers should also be sent back to the Edge. The report from the Edge cache contains all this information.

Origin Cache While the Edge trace already contains the hit/miss status at Origin servers, it does not provide details about communication between the Origin servers and the Backend. Therefore, we also have each Origin host report sampled events to Scribe when a request to the Backend is completed.

To ensure that the same photos are sampled in all three traces, our sampling strategy is based on hashing: we sample a tunable percentage of events by means of a deterministic test on the photoId. We explore this further in Section 3.3.3.

Scribe aggregates logs and loads them into Hive [82], Facebook's data warehouse. Scripts then perform statistical analyses yielding the graphs shown below.

3.3.2 Correlating Requests

By correlating traces between the different layers of the stack we accomplish several goals. First, we can ask what percentage of requests result in cache hits within the client browser. Additionally, we can study the paths taken by individual requests as they work their way down the stack. Our task would be trivial if we could add unique request-IDs to every photo request at the browser and then piggyback that information on the request as it travels along the stack, such an approach would be disruptive to the existing Facebook code base. This forces us to detect correlations in ways that are sometimes indirect, and that are accurate but not always perfectly so.

The first challenge arises in the client, where the detection of client-side cache hits is complicated by a technicality: although we do know which URLs are accessed, if a photo request is served by the browser cache our Javascript instrumentation has no way to determine that this was the case. For example, we can't infer that a local cache hit occured by measuring the time delay between photo fetch and completion: some clients are so close to Edge Caches that an Edge response could be faster than the local disk. Accordingly, we infer the aggregated cache performance for client object requests by comparing the number of requests seen at the browser with the number seen in the Edge for the same URL.

To determine the geographical flow between clients and PoPs, we correlate browser traces and Edge traces on a per request basis. If a client requests a URL and then an Edge Cache receives a request for that URL from the client's IP address, then we assume a miss in the browser cache triggered an Edge request. If the client issues multiple requests for a URL in a short time period and there is one request to an Edge Cache, then we assume the first request was a miss at browser but all subsequent requests were hits.

Correlating Backend-served requests in the Edge trace with requests between the Origin and Backend layers is relatively easy because they have a one-to-one mapping. If a request for a URL is satisfied after an Origin miss, and a request for the same URL occurs between the same Origin host and some Backend server, then we assume they are correlated. If the same URL causes multiple misses at the same Origin host, we align the requests with Origin requests to the Backend in timestamp order.

3.3.3 Sampling Bias

To avoid affecting performance, we sample requests instead of logging them all. Two sampling strategies were considered: (1) sampling requests randomly, (2) sampling focused on some subset of photos selected by a deterministic test on photoId. We chose the latter for two reasons:

- **Fair coverage of unpopular photos:** Sampling based on the photo identifier enables us to avoid bias in favor of transiently popular items. A biased trace could lead to inflated cache performance results because popular items are likely stored in cache.
- **Cross stack analysis:** By using a single deterministic sampling rule that depends only on the unique photoId, we can capture and correlate events occurring at different layers.

A potential disadvantage of this approach is that because photo-access workload is Zipfian, a random hashing scheme could collect different proportions of photos from different popularity levels. This can cause the estimated cache performance to be inflated or deflated, reflecting an overly high or low coverage of popular objects. To quantify the degree of bias in our traces, we further downsampled our trace to two separate data sets,

each of which covers 10% of our original photoIds. While one set inflates the hit ratios at browser, Edge and Origin caches by 3.6%, 2% and 0.4%, the other set deflates the hit ratios at browser and Edge caches 0.5% and 4.3%, resp. Overall, the browser and Edge cache performance are more sensitive to workload selection based on photoIds than the Origin. Comparing to Facebook's live monitoring data, which has a higher sampling ratio but lower sampling duration, our reported Edge hit ratio is lower by about 5% and our Origin hit ratio is about the same. We concluded that our sampling scheme is reasonably unbiased.

3.3.4 Privacy Preservation

We took a series of steps to preserve the privacy of Facebook users. First, all raw data collected for this study was kept within the Facebook data warehouse (which lives behind a company firewall) and deleted within 90 days. Second, our data collection logic and analysis pipelines were heavily reviewed by Facebook employees to ensure compliance with Facebook privacy commitments. Our analysis does not access image contents or users profiles (however, we do sample some meta-information: photo size, age and the owner's number of followers). Finally, as noted earlier, our data collection scheme is randomized and based on photoId, not user-id; as such, it only yields aggregated statistics for a cut across the total set of photos accessed during our study period.

3.4 Workload Characteristics

Our analysis examines more than 70 TB of data, all corresponding to client-initiated requests that traversed the Facebook photo-serving stack during a one-month sampling

	Inside browser	Edge caches	Origin cache	Backend (Haystack)
Photo requests	77,155,557	26,589,471	11,160,180	7,606,375
Hits	50,566,086	15,429,291	3,553,805	7,606,375
% of traffic served	65.5%	20.0%	4.6%	9.9%
Hit ratio	65.5%	58.0%	31.8%	N/A
Photos w/o size	1,384,453	1,301,972	1,300,476	1,295,938
Photos w/ size	2,678,443	2,496,512	2,484,155	1,531,339
Users	13,197,196	N/A	N/A	N/A
Client IPs	12,341,785	11,083,418	1,193	1,643
Client geolocations	24,297	23,065	24	4
Bytes transferred	N/A	492 GB	251 GB	457 GB (187 GB post-resize)

Table 3.1: Workload characteristics: broken down by different layers across the photo-serving stack where traffic was observed; *Client IPs* refers to the number of distinct IP addresses identified on the requester side at each layer, and *Client geolocations* refers to the number of distinct geographical regions to which those IPs map.

period. Table 3.1 shows summary statistics for our trace. Our trace includes over 77M requests from 13.2M user browsers for more than 1.3M unique photos. Our analysis begins with a high level characterization of the trace, and then dives deeper in the sections that follow.

Table 3.1 gives the number of requests and hits at each successive layer. Of the 77.2M browser requests, 50.6M are satisfied by browser caches (65.5%), 15.4M by the Edge Caches (20.0%), 3.6M by the Origin cache (4.6%), and 7.6M by the Backend (9.9%). There is an enormous working set and the photo access distribution is long-tailed with a significant percentage of accesses are directed to low-popularity photos. Looking next at bytes being transferred at different layers, we see that among 492.2GB of photo traffic being delivered to the client, 492.2 – 250.6 = 241.6GB were served by Edge caches, 250.6 – 187.2 = 63.4GB were served by the Origin and 187.2GB were derived from Backend fetches, which corresponds to over 456GB of traffic between the Origin and Backend before resizing.

This table also gives the hit ratio at each caching layer. The 65.5% hit ratio at client

browser caches provides significant traffic sheltering to Facebook's infrastructure. Without the browser caches, requests to the Edge Caches would approximately triple. The Edge Caches have a 58.0% hit ratio and this also provides significant traffic sheltering to downstream infrastructure: if the Edge Caches were removed, requests to the Origin Cache and the bandwidth required from it would more than double. Although the 31.8% hit ratio achieved by the Origin Cache is the lowest among the caches present in the Facebook stack, any hits that do occur at this level reduce costs in the storage layer and eliminate backend network cost, justifying deployment of a cache at this layer.

Recall that each size of a photo is a distinct object for caching purposes. The *Photos w/o size* row ignores the size distinctions and presents the number of distinct underlying photos being requested. The number of distinct photos requests at each tier remains relatively constant, about 1.3M. This agrees with our intuition about caches: they are heavily populated with popular content represented at various sizes, but still comprise just a small percentage of the unique photos accessed in any period. For the large numbers of unpopular photos, cache misses are common. The *Photos w/ size* row breaks these figures down, showing how many photos are requested at each layer, but treating each distinct size of an image as a separate photo. While the number decreases as we traverse the stack, the biggest change occurs in the Origin tier, suggesting that requests for new photo sizes are a source of misses. The Haystack Backend maintains each photo at four commonly-requested sizes, which helps explain why the count seen in the last column can exceed the number of unique photos accessed: for requests corresponding to these four sizes, there is no need to undertake a (costly) resizing computation.

The size distribution of transferred photos depends upon the location at which traffic is observed. Figure 3.2 illustrates the cumulative distribution of object size transferred before and after going through the Origin Cache for all Backend fetches. After photos

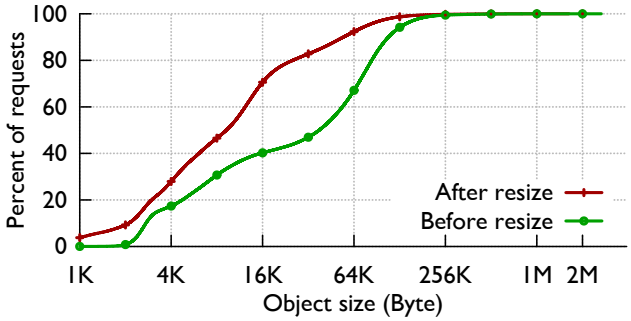


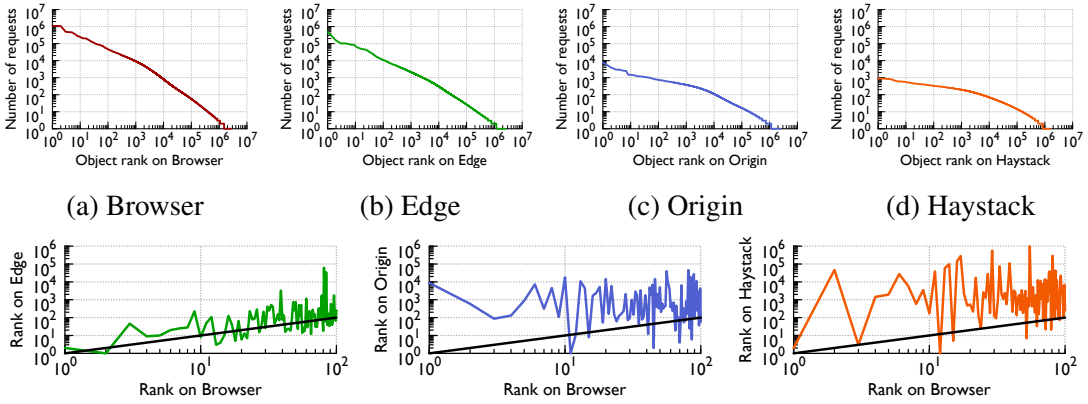
Figure 3.2: Cumulative distribution function (CDF) on object size being transferred through the Origin cache.

are resized, the percentage of transferred objects smaller than 32KB increases from 47% to over 80%.

The rows labeled *Client IPs* and *Client geolocations* in Table 3.1 offer measurements of coverage of our overall instrumentation stack. For example, we see that more than 12 million distinct client IP addresses covering over 24 thousand geolocations (cities or towns) used the system, that 1,193 distinct Facebook Edge caches were tracked, etc. As we move from left to right through the stack we see traffic aggregate from massive numbers of clients to a moderate scale of Edge regions and finally to a small number of data centers. Section 3.5 undertakes a detailed analysis of geolocation phenomena.

3.4.1 Popularity Distribution

A natural way to quantify object popularity is by tracking the number of repeated requests for each photo. For Haystack we consider each stored common sized photo as an object. For other layers we treat each resized variant as an object distinct from the underlying photo. By knowing the number of requests for each object, we can then explore the significance of object popularity in determining Facebook's caching performance. Prior studies of web traffic found that object popularity follows a Zipfian



(a) Browser (b) Edge (c) Origin (d) Haystack

(e) Rank shift from Browser to Edge (f) Rank shift from Browser to Origin (g) Rank shift from Browser to Haystack

Figure 3.3: Popularity distribution. Top: Number of requests to unique photos at each layer, ordered from the most popular to the least. Bottom: a comparison of popularity of items in each layer to popularity in the client browser, with the exact match shown as a straight line. Shifting popularity rankings are thus evident as spikes. Notice in (a)-(d) that as we move deeper into the stack, these distributions flatten in a significant way.

distribution [19]. Our study of browser access patterns supports this finding. However, at deeper levels of the photo stack, the distribution flattens, remaining mostly Zipf-like (with decreasing Zipf-coefficient α at each level), but increasingly distorted at the head and tail. By the time we reach the Haystack Backend, the distribution more closely resembles a *stretched exponential* distribution [40].

Figures 3.3a, 3.3b, 3.3c and 3.3d show the number of requests to each unique photo blob as measured at different layers, ordered by popularity rank in a log-log scale. Because each layer absorbs requests to some subset of items, the rank of each blob can change if popularity is recomputed layer by layer. To capture this effect visually, we plotted the rank shift, comparing popularity in the browser ranking to that seen in the Edge (Figure 3.3e), in the Origin tier (Figure 3.3f) and in Haystack (Figure 3.3g). In these graphs, the x-axis is the rank of a particular photo blob as ordered on browsers, while the y-axis gives the rank on the indicated layer for that same photo object. The

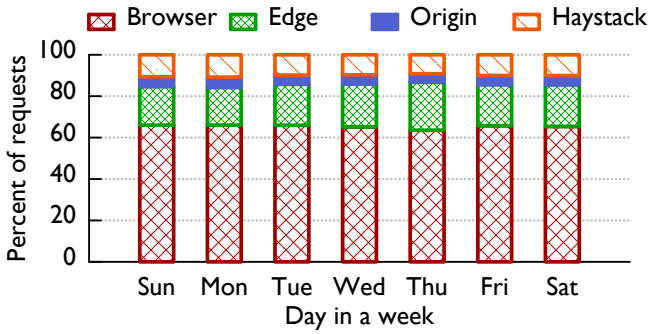
type of blob is decided by the indicated layer. Had there been no rank shift, these graphs would match the straight black line seen in the background.

As seen in these plots, item popularity distributions at all layers are approximately Zipfian (Figures 3.3a-3.3d). However, level by level, item popularities shift, especially for the most popular 100 photo blobs in the Edge's popularity ranking. For example, when we look at the rank shift between browser and Edge (Figure 3.3e), where 3 top-10 objects dropped out of the highest-popularity ranking and a substantial fraction of the 10th-100th most popular objects dropped to around 1000th and even 10000th on the Edge ("upward" spikes correspond to items that were more popular in the browser ranking than in the Edge ranking).

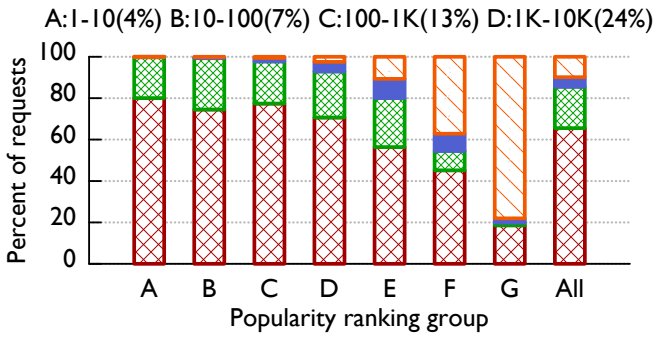
As traffic tunnels deeper into the stack and reaches first the Origin Cache and then Haystack, millions of requests are served by each caching layer, hence the number of requests for popular images is steadily reduced. This explains why the distributions seen on the Edge, Origin and Haystack remain approximately Zipfian, but the Zipf coefficient, α, becomes smaller: the stream is becoming steadily less cacheable. Yet certain items are still being cached effectively, as seen by the dramatic popularity-rank shifts as we progress through the stack.

3.4.2 Hit Ratio

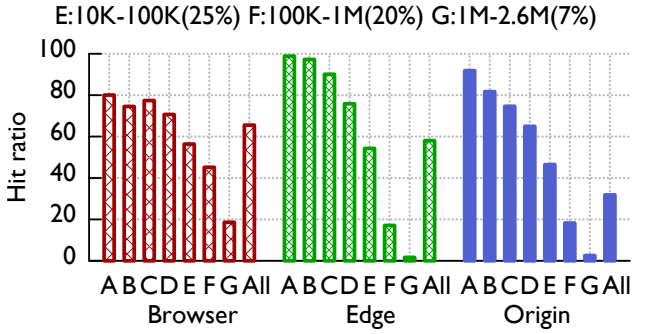
Given insight into the popularity distribution for distinct photo blobs, we can relate popularity to cache hit ratio performance as a way to explore the question posed earlier: *To what extent does photo blob popularity shape cache hit ratios?* Figure 3.4a illustrates the traffic share in terms of percent of client's requests served by each layer during a period of approximately one week. Client browsers resolved ~65% of the traffic from



(a) Traffic share for a week



(b) Traffic share for popularity groups



(c) Hit ratio for popularity groups

Figure 3.4: Traffic distribution. Percent of photo requests served by each layer, (a) aggregated daily for a week; (b) binned by image popularity rank on a single day. For (b), 1-10 represent the 10 most popular photos, 10-100 the 90 next most popular, etc. (c) shows the hit ratios for each cache layer binned by the same popularity group, along with each group's traffic share.

the local browser cache, the Edge cache served ~20%, the Origin tier ~5%, and Haystack handled the remaining ~10%. Although obtained differently, these statistics are consistent with the aggregated results we reported in Table 3.1.

Figure 3.4b breaks down the traffic served by each layer into image-popularity groups. We assign each photo blob a popularity rank based on the number of requests in our trace. The most popular photo blob has rank 1 and the least popular blob has rank over 2.6M. We then bin items by popularity, using logarithmically-increasing bin sizes. The figure shows that the browser cache and Edge cache served more than 89% of requests for the hundred-thousand most popular images (groups A-E). As photo blobs become less popular (groups F then G) they are less likely to be resident in cache and thus a higher percentage of requests are satisfied by the Haystack Backend. In particular, we see that Haystack served almost 80% of requests for the least popular group (G). The Origin Cache also shelters the Backend from a significant amount of traffic, and this sheltering is especially effective for blobs in the middle popularity groups (D, E and F), which are not popular enough to be retained in the Edge cache.

Figure 3.4c illustrates the hit ratios binned by the same popularity groups for each cache layer. It also shows the percent of requests to each popularity group. One interesting result is the dramatically higher hit ratios for the Edge and Origin layers than the browser cache layer for popular photos (groups A-B). The explanation is straightforward. Browser caches can only serve photos that this particular client has previously downloaded, while the Edge and Origin caches are shared across all clients and can serve photos any client has previously downloaded. The reverse is true for unpopular photos (groups E-G). They have low hit ratios in the shared caches because they are quickly evicted for more generally popular content, but remain in the individual browser caches, which see traffic from only a single user.

Looking closely at the hit ratios, it is at first counterintuitive that the browser cache has a lower hit ratio for group B than the next less popular photo group C. The likely reason is that many photo blobs in this group are "viral," in the sense that large numbers

Popularity group	# Requests	# Unique IPs	Req/IP ratio
A	5120408	665576	7.7
B	8313854	1530839	5.4
C	15497215	2302258	6.7

Table 3.2: Access statistics for selected groups. "Viral" photos are accessed by massive numbers of clients, rather than accessed many times by few clients, so browser caching is of only limited utility.

of distinct clients are accessing them concurrently. Table 3.2 confirms this by relating the number of requests, the number of distinct IP addresses, and the ratio between these two for the top 3 popularity groups. As we can see, the ratio between the number of requests and the number of IP addresses for group B is lower than the more popular group A and less popular group C. We conclude that although many clients will access "viral" content once, having done so they are unlikely to subsequently revisit that content. On the other hand, a large set of photo blobs are repeatedly visited by the same group of users. This demonstrates that the Edge cache and browser cache complement one another in serving these two categories of popular images, jointly accounting for well over 90% of requests in popularity groups A, B and C of Figure 3.4b.

3.5 Geographic Traffic Distribution

This section explores the geographical patterns in request flows. We analyze traffic between clients and Edge Caches, how traffic is routed between the Edge Caches and Origin Cache, and how Backend requests are routed. Interestingly (and somewhat surprisingly), we find significant levels of cross-country routing at all layers.

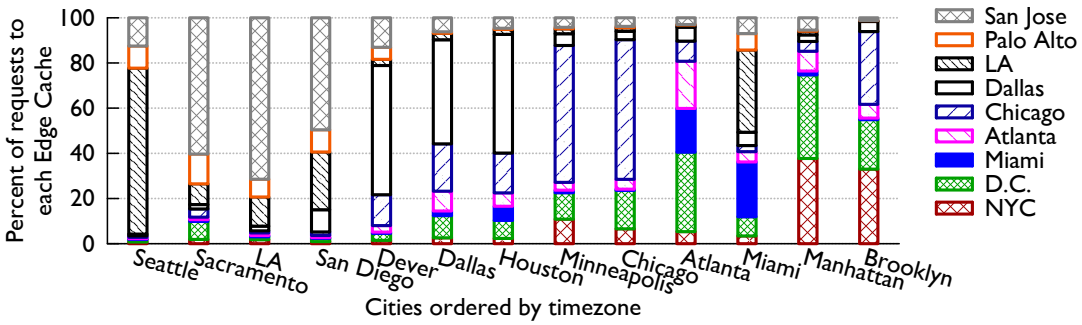


Figure 3.5: Traffic share from 13 large cities to Edge Caches (identified in legend at right).

3.5.1 Client To Edge Cache Traffic

We created a linked data set that traces activities for photo requests from selected cities to US-based Edge Caches. We selected thirteen US-based cities and nine Edge Caches, all heavily loaded during the period of our study. Figure 3.5 shows the percentage of requests from each city that was directed to each of the Edge Caches. Timezones are used to order cities (left is West) and Edge Caches (top is West).

Notice that each city we examine is served by all nine Edge Caches, even though in many cases this includes Edge Caches located across the country that are accessible only at relatively high latency. Indeed, while every Edge Cache receives a majority of its requests from geographically nearby cities, the largest share does not necessarily go to the nearest neighbor. For example, fewer Atlanta requests are served by the Atlanta Edge Cache than by the D.C. Edge Cache. Miami is another interesting case: Its traffic was distributed among several Edge Caches, with 50% shipped west and handled in San Jose, Palo Alto and LA and only 24% handled in Miami.

The reason behind this geographical diversity is a routing policy based on a combination of latency, Edge Cache capacity and ISP peering cost, none of which necessarily translates to physical locality. When a client request is received, the Facebook DNS

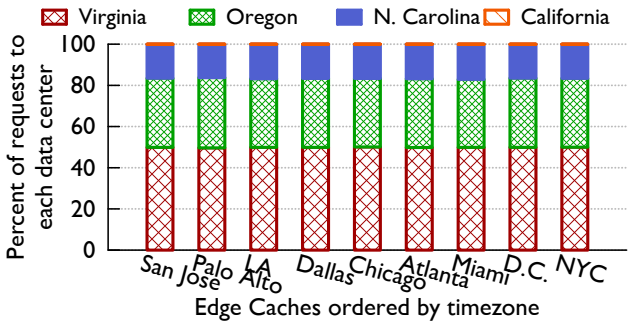


Figure 3.6: Traffic from major Edge Caches to the data centers that comprise the Origin Cache.

server computes a weighted value for each Edge candidate, based on the latency, current traffic, and traffic cost, then picks the best option. The peering costs depend heavily on the ISP peering agreements for each Edge Cache, and, for historical reasons, the two oldest Edge Caches in San Jose and D.C. have especially favorable peering quality with respect to the ISPs hosting Facebook users. This increases the value of San Jose and D.C. compared to the other Edge Caches, even for far-away clients.

A side effect of Facebook's Edge Cache assignment policy is that a client may shift from Edge Cache to Edge Cache if multiple candidates have similar values, especially when latency varies throughout the day as network dynamics evolve. We examined the percentage of clients served by a given number of Edge Caches in our trace: 0.9% of clients are served by 4 or more Edge Caches, 3.6% of clients are served by 3 or more Edge Caches, and 17.5% of clients are served by 2 or more Edge Caches. Client redirection reduces the Edge cache hit ratio because every Edge Cache reassignment brings the potential for new cold cache misses. In Section 3.7, we discuss potential improvement from collaborative caching.

3.5.2 Edge Cache to Origin Cache Traffic

Currently Facebook serves user-uploaded photos at four regional data centers in the United States. Two are on the East Coast (in Virginia and North Carolina) and two others are on the West Coast (in Oregon and California). In addition to hosting Haystack Backend clusters, these data centers comprise the Origin Cache configured to handle requests coming from various Edge Caches. Whenever there is an Edge Cache miss, the Edge Cache will contact a data center based on a consistent hashed value of that photo. In contrast with the Edge Caches, all Origin Cache servers are treated as a single unit and the traffic flow is purely based on content, not locality.

Figure 3.6 shows the share of requests from nine Edge Caches to the four Origin Cache data centers. The percentage of traffic served by each data center on behalf of each Edge Cache is nearly constant, reaffirming the effects of consistent hashing. One noticeable exception, California, was being decommissioned at the time of our analysis and not absorbing much Backend traffic.

3.5.3 Cross-Region Traffic at Backend

In an ideal scenario, when a request reaches a data center we would expect it to remain within that data center: the Origin Cache server will fetch the photo from the Backend in the event of a miss, and a local fetch would minimize latency. But two cases can arise that break this common pattern:

- **Misdirected resizing traffic:** Facebook continuously migrates Backend data, both for maintenance and to ensure that there are adequate numbers of backup copies of each item. Continuously modifying routing policy to keep it tightly

Origin Cache Region	Backend Region		
	Virginia	North Carolina	Oregon
Virginia	99.885%	0.049%	0.066%
North Carolina	0.337%	99.645%	0.018%
Oregon	0.149%	0.013%	99.838%
California	24.760%	13.778%	61.462%

Table 3.3: Origin Cache to Backend traffic. Most Origin traffic stays within the same data center, with an exception for Origin in California, where the data center was being decommissioned during the period of our study.

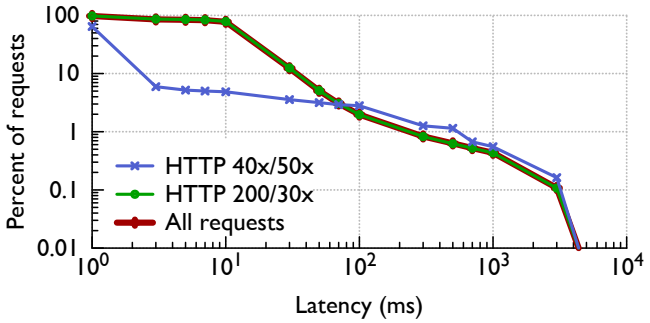


Figure 3.7: Complementary cumulative distribution function (CCDF) on latency of requests from Origin Cache servers to the Backend.

aligned with replica location is not feasible, so the system tolerates some slack, which manifests in occasional less-than-ideal routing.

- **Failed local fetch:** Failures are common at scale, and thus the Backend server holding some local replica of a desired image may be offline or overloaded. When a request from an Origin Cache server to its nearby Backend fails to fetch a photo quickly, the Origin Cache server will pick a remote alternative.

Table 3.3 summarizes the traffic retention statistics for each data center. More than 99.8% of requests were routed to a data center within the region of the originating Origin Cache, while about 0.2% of traffic travels over long distances. This latter category was dominated by traffic sent from California, which is also the least active data center region in Figure 3.6 for Edge Caches. As noted earlier, this data center was being decommissioned during the period of our study.

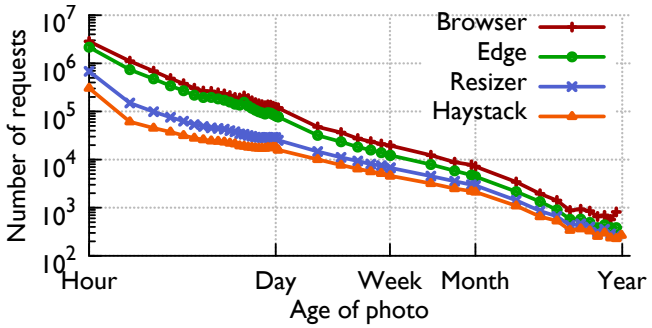
Figure 3.7 examines the latency of traffic between the Origin Cache servers and the Backend, with lines for successful requests (HTTP error code 200/30x), failed requests (HTTP error code 40x/50x), and all requests. While successful accesses dominate, more than 1% of requests failed. Most requests are completed within tens of milliseconds. Beyond that range, latency curves have two inflection points at 100ms and 3s, corresponding to the minimum delays incurred for cross-country traffic between eastern and western regions, and maximum timeouts currently set for cross-country retries. When a successful re-request follows a failed request, the latency is aggregated from the start of the first request.

3.6 Social-Network Analysis

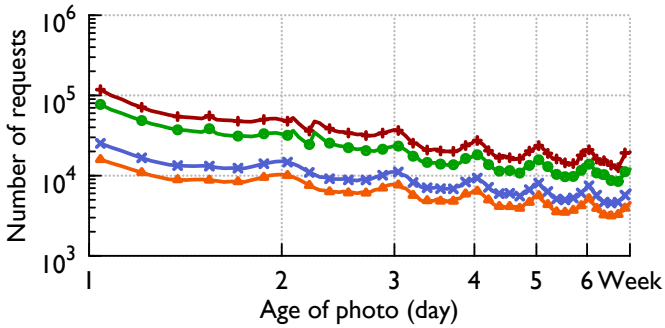
This section explores the relationship between photo requests and various kinds of photo meta-information. We studied two properties that intuitively should be strongly associated with photo traffic: the age of photos and the number of Facebook followers associated with the owner.

3.6.1 Content Age Effect

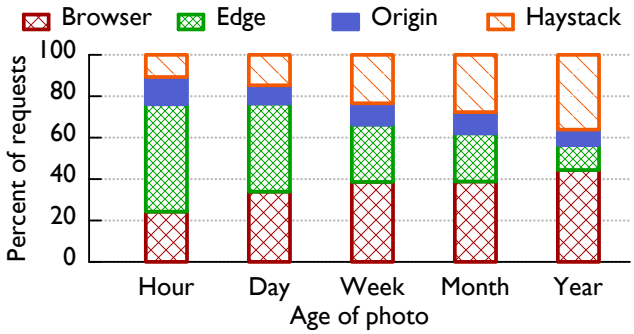
It is generally assumed that new content will draw attention and hence account for the majority of traffic seen within the blob-serving stack. Our data set permits us to evaluate such hypotheses for the Facebook image hierarchy by linking the traces collected from different layers to the meta-information available in Facebook's photo database. We carried out this analysis, categorizing requests for images by the age of the target content, then looking at the way this information varies at each layer of the stack. Photo age



(a) Age spans from 1 hour to 1 year



(b) Age spans from 1 day to 1 week



(c) Traffic share for non-profile photos by age

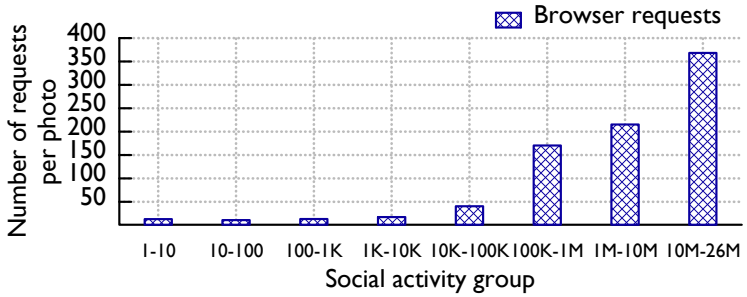
Figure 3.8: Traffic popularity and requests served by layer for photos at different age. The number of requests to each image, categorized by age of requested photos in hours, broken down at every layer across the stack.

(in hours) was determined by subtracting the photo creation time from the request time. Thus, even a photo uploaded the same day will have associated requests sorted into 24 hourly categories. This analysis excludes profile photos because Facebook's internal storage procedures for them precludes determining their ago precisely.

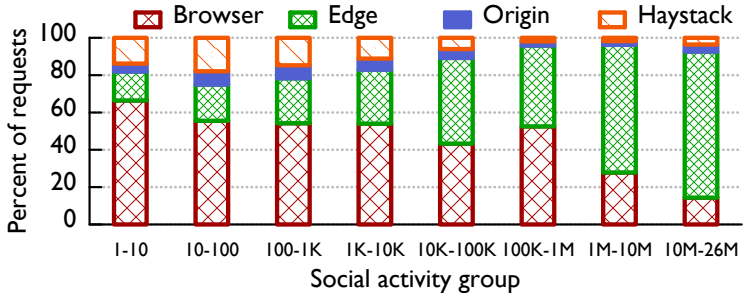
Figure 3.8 plots the number of requests at each layer for photos of different ages. In Figure 3.8a, we consider a range of ages from 1 hour to 1 year. As content ages, the associated traffic diminishes at every layer; the relationship is nearly linear when plotted on a log-log scale. Figure 3.8b zooms into the mid-range age scales, focusing on a week. We see a noticeable daily traffic fluctuation. We traced this to a fluctuation in photo creation time, determining that users create and upload greater numbers of photos during certain periods of the day. This creation-time effect carries through to induce the striking photo-access-by-age pattern observed for smaller ages.

Our analysis reveals that traffic differences between caches deployed close to clients (browser, Edge Cache) and storage Backend (including the Origin Cache) are more pronounced for young photos than for old ones. This matches intuition: fresh content is popular and hence tends to be effectively cached throughout the image serving hierarchy, resulting in higher cache hit ratios. Figure 3.8c clearly exhibits this pattern. The age-based popularity decay of photos seen in Figure 3.8a is nearly Pareto, suggesting that an age-based cache replacement algorithm could be effective.

We should note that although our traces include accesses to profile photos, and we used them in all other analyses, we were forced to exclude profile photos for this age-analysis. The issue relates to a quirk of the Facebook architecture: when a user changes his or her profile photo, Facebook creates a new profile object but reuses the same name as for the previous versions. Profile objects can be distinguished by looking at the ownerId, which Facebook sets to the underlying photoId, but we can not determine the time of creation. None of our other analyses are impacted, but we were forced to exclude profile objects in our age analysis. The effect is to slightly reduce the computed traffic share for caches close to clients, especially in the categories associated with young and popular photos.



(a) Client requests per photo



(b) Traffic share for social activity groups

Figure 3.9: Photo popularity organized owner popularity. (a) Requests per photo categorized by the number of followers for the photo's owner. (b) Traffic distribution by layer for different social activity groups.

3.6.2 Social Effects

Intuitively, we expect that the more friends a photo owner has, the more likely the photo is to be accessed. We observed this phenomenon in our study, but only when we condition on owner type. We binned owners by the number of followers (friends for normal users, fans for public page owners), creating "popularity groups", and graphed photo requests by their owners' groups, yielding the data seen in Figure 3.9. For each photo request, the photo owner's friend count was fetched on the day when the access happened, thus requests for one photo may be split into multiple groups when an owner's popularity changes. We include profile photos in this analysis.

Figure 3.9a graphs the number of requests for each photo against the photo owner's popularity group. Most Facebook users have fewer than 1000 friends, and for that range

the number of requests for each photo is almost constant. For public page owners who can have thousands or millions of fans, each photo has a significantly higher number of requests, determined by the size of the fan base. Figure 3.9b further breaks down the traffic distribution at each layer of the photo-serving stack for different social activity groups. For normal users with fewer than 1000 followers (friends), the caches absorb ~80% of the requests for their photos; but for public page owners, more followers (fans) drives higher percentages of traffic being absorbed by caches. However, browser caches tend to have lower hit ratios for owners with more than 1 million followers. This is because these photos fall into the "viral" category discussed earlier in Section 3.4.

3.7 Potential Improvements

This section closely examines Browser, Edge, and Origin Cache performance. We use simulation to evaluate the effect of different cache sizes, algorithms, and strategies.

3.7.1 Browser Cache

Figure 3.10 shows the aggregated hit ratio we observed for different groups of clients. The "all" group includes all clients and had a aggregated hit ratio of 65.5%. This is much higher than the browser cache statistics published by the Chrome browser development team for general content: they saw hit ratios with a Gaussian distribution around a median of 35% for unfilled caches and 45% for filled caches [23].

The figure also breaks down hit ratios based on the observed activity level of clients, i.e., how many entries are in our log for them. The least active group with 1-10 logged requests saw a 39.2% hit ratio, while a more active group with 1K-10K logged requests

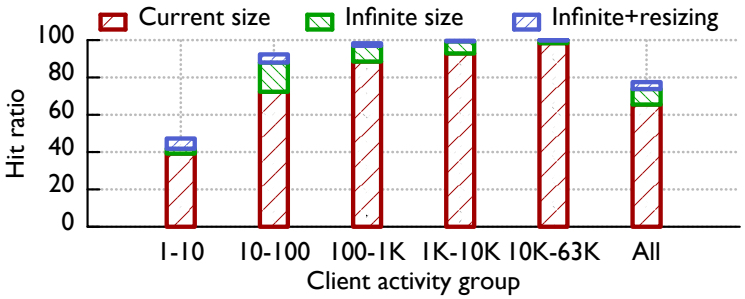


Figure 3.10: Measured, ideal, and resize-enabled hit ratios for clients grouped by activity. 1-10 groups clients with ≤ 10 requests, 10-100 groups those reporting 11 to 100 requests, etc. *All* groups all clients.

saw a 92.9% hit ratio. The higher hit ratio for more active clients matches our intuition: highly active clients are more likely to access repeated content than less active clients, and thus their browser caches can achieve a higher hit ratio.

Browser Cache Simulation Using our trace to drive a simulation study, we can pose *what-if* questions. In Figure 3.10 we illustrate one example of the insights gained in this manner. We investigate what the client browser hit ratios would have been with an infinite cache size. We use the first 25% of our month-long trace to warm the cache and then evaluate using the remaining 75% of the trace. The infinite cache size results distinguish between cold (compulsory) misses for never-before-seen content and capacity misses, which never happen in an infinite cache. The infinite size cache bar thus gives an upper bound on the performance improvements that could be gained by increasing the cache size or improving the cache replacement policy. For most client activity groups this potential gain is significant, but the least active client group is an interesting outlier. These very inactive clients would see little benefit from larger or improved caches: an unbounded cache improved their hit ratio by 2.6% to slightly over 41.8%.

We also simulated the effect of moving some resizing to the client: clients with a cached full-size image resize that object rather than fetching the required image size.

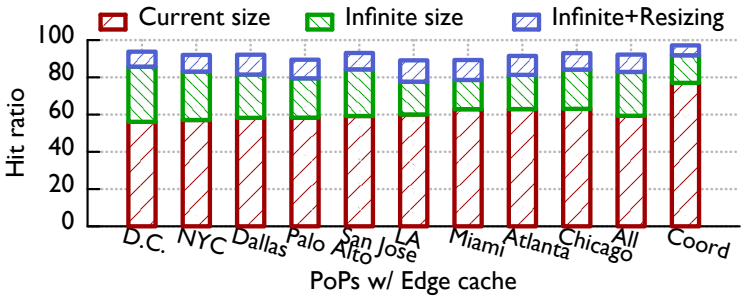


Figure 3.11: Measured, ideal, and resize-enabled hit ratios for the nine largest Edge Caches. *All* is the aggregated hit ratio for all regions. *Coord* gives the results for a hypothetical collaborative Edge Cache.

While client-side resizing does not result in large improvements in hit ratio for most client groups, it does provide a significant 5.5% improvement even relative to an unbounded cache for the least active clients.

3.7.2 Edge Cache

To investigate Edge cache performance at a finer granularity, we analyzed the hit ratio for nine heavily used Edge Caches. Figure 3.11 illustrates the actual hit ratio observed at each Edge Cache, a value aggregated across all regions, denoted "All", and a value for a hypothetical collaborative cache that combines all Edge Caches into a single Edge Cache. (We defer further discussion of the collaborative cache until later in this subsection.) We also estimated the highest possible hit ratio for perfect Edge Caches by replaying access logs and assuming an infinite cache warmed by the first 25% of our month-long trace. We then further enhanced the hypothetical perfect Edge Caches with the ability to resize images. The results are stacked in Figure 3.11, with the actual value below and the simulated ideal performance contributing the upper portion of each bar.

The current hit ratios range from 56.1% for D.C. to 63.1% in Chicago. The upper bound on improvement, infinite size caches, has hit ratios from 77.7% in LA to 85.8% in

Algo.	Description
FIFO	A first-in-first-out queue is used for cache eviction. This is the algorithm Facebook currently uses.
LRU	A priority queue ordered by last-access time is used for cache eviction.
LFU	A priority queue ordered first by number of hits and then by last-access time is used for cache eviction.
S4LRU	Quadruply-segmented LRU. Four queues are maintained at levels 0 to 3. On a cache miss, the item is inserted at the head of queue 0. On a cache hit, the item is moved to the head of the next higher queue (items in queue 3 move to the head of queue 3). Each queue is allocated $1/4$ of the total cache size and items are evicted from the tail of a queue to the head of the next lower queue to maintain the size invariants. Items evicted from queue 0 are evicted from the cache.
Clairvoyant	A priority queue ordered by next-access time is used for cache eviction. (Requires knowledge of the future.)
Infinite	No object is ever evicted from the cache. (Requires a cache of infinite size.)

Table 3.4: Descriptions of the simulated caching algorithms.

D.C.. While the current hit ratios represent significant traffic sheltering and bandwidth reduction, the much higher ratios for infinite caches demonstrate there is much room for improvement. The even higher hit ratios for infinite caches that can resize photos makes this point even clearer: hit ratios could potentially be improved to be as high as 89.1% in LA, and to 93.8% in D.C..

Edge Cache Simulation Given the possibility of increases as high as 40% in hit ratios, we ran a number of *what-if* simulations. Figures 3.12a and 3.12b explores the effect of different cache algorithms and cache sizes for the San Jose Edge Cache. We use San Jose here because it is the median in current Edge Cache hit ratios and the approximate visual median graph of all nine examined Edge Caches. The horizontal gray bar on the graph corresponds to the observed hit ratio for San Jose, 59.2%. We label the x-coordinate of the intersection between that observed hit ratio line and the FIFO simulation line, which is the current caching algorithm in use at Edge Caches, as size x. This is our approximation of the current size of the cache at San Jose.

The different cache algorithms we explored are explained briefly in Table 3.4. We

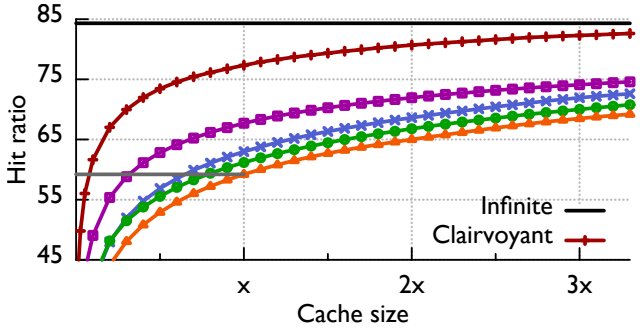
first examine the results for object-hit ratio. Our results demonstrate that more sophisticated algorithms yield significant improvements over the current FIFO algorithm: 2.0% from LFU, 3.6% from LRU, and 8.5% from S4LRU. Each of these improvements yields a reduction in downstream requests. For instance, the 8.5% improvement in hit ratio from S4LRU yields a 20.8% reduction in downstream requests.

The performance of the Clairvoyant algorithm demonstrates that the infinite-size-cache hit ratio of 84.3% is unachievable at the current cache size. Instead, an almost-theoretically-perfect algorithm could only achieve a 77.3% hit ratio.[1] This hit ratio still represents a very large potential increase of 18.1% in hit ratio over the current FIFO algorithm, which corresponds to a 44.4% decrease in downstream requests. The large gap between the best algorithm we tested, S4LRU, and the Clairvoyant algorithm demonstrates there may be ample gains available to still-cleverer algorithms.

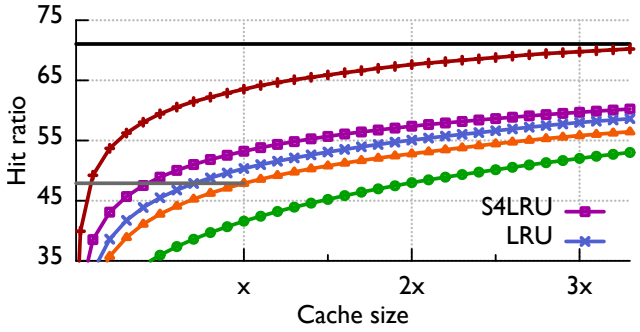
The object-hit ratios correspond to the success of a cache in sheltering traffic from downstream layers, i.e., decreasing the number of requests (and ultimately disk-based IO operations). For Facebook, the main goal of Edge Caches is not traffic sheltering, but bandwidth reduction. Figure 3.12b shows byte-hit ratios given different cache sizes. These results, while slightly lower, mostly mirror the object-hit ratios. LFU is a notable exception, with a byte-hit ratio below that of FIFO. This indicates that LFU would not be an improvement for Facebook because even though it can provide some traffic sheltering at the Edge, it increases bandwidth consumption. S4LRU is again the best of the tested algorithms with an increase of 5.3% in byte-hit ratio at size x, which translates to a 10% decrease in Origin-to-Edge bandwidth.

Figure 3.12 also demonstrates the effect of different cache sizes. Increasing the

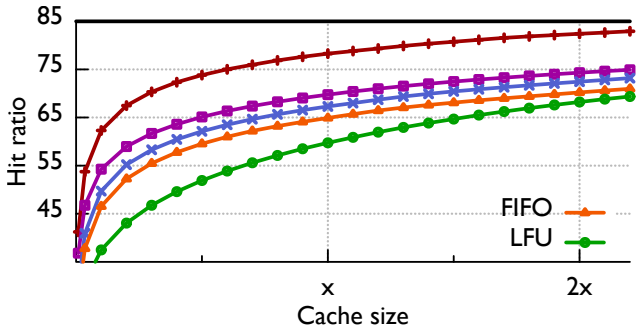
[1]The "Clairvoyant" algorithm is not theoretically perfect because it does not take object size into account. It will choose to store an object of size $2x$ next accessed at time t over storing 2 objects of size x next accessed at times $t + 1$ and $t + 2$.



(a) Object-Hit Ratio at San Jose



(b) Byte-Hit Ratio at San Jose



(c) Byte-Hit Ratio for a Collaborative Edge

Figure 3.12: Simulation of Edge Caches with different cache algorithms and sizes. The object-hit ratio and byte-hit ratio are shown for the San Jose Edge Cache in (a) and (b), respectively. The byte-hit ratio for a collaborative Edge Cache is given in (c). The gray bar gives the observed hit ratio and size x approximates the current size of the cache.

size of the cache is also an effective way to improve hit ratios: doubling the cache size increases the object-hit ratio of the FIFO algorithm by 5.8%, the LFU algorithm by 5.6%, the LRU algorithm by 5.7%, and the S4LRU algorithm by 4.3%. Similarly,

it increases the byte-hit ratios of the FIFO algorithm by 4.8%, the LFU algorithm by 6.4%, the LRU algorithm by 4.8%, and the S4LRU algorithm by 4.2%.

Combining the analysis of different cache algorithms and sizes yields even more dramatic results. There is an inflection point for each algorithm at a cache size smaller than x. This translates to higher-performing algorithms being able to achieve the current object-hit ratio at much smaller cache sizes: LFU at $0.8x$, LRU at $0.65x$, and S4LRU at $0.35x$. The results are similar for the size needed to achieve the current byte-hit ratio: LRU at $0.7x$ and S4LRU at $0.3x$. These results provide a major insight to inform future static-content caches: a small investment in Edge Caches with a reasonably sophisticated algorithm can yield major reductions in traffic. Further, the smaller a cache, the greater the choice of algorithm matters.

Collaborative Edge Cache We also simulated a collaborative Edge Cache that combines all current Edge Caches into a single logical cache. Our motivation for this *what-if* scenario is twofold. First, in the current Edge Cache design, a popular photo may be stored at every Edge Cache. A collaborative Edge Cache would only store that photo once, leaving it with extra space for many more photos. Second, as we showed in Section 3.5, many clients are redirected between Edge Caches, resulting in cold misses that would be avoided in a collaborative cache. Of course, this hypothetical collaborative Edge Cache might not be ultimately economical because it would incur greater peering costs than the current system and would likely increase client-photo-load latency.

Figure 3.12c gives the byte-hit ratio for different cache algorithms and sizes for a collaborative Edge Cache. The size x in this graph is the sum of the estimated cache size we found by finding the intersection of observed hit ratio and FIFO simulation hit ratio for each of the nine Edge Caches. At the current cache sizes, the improvement in hit

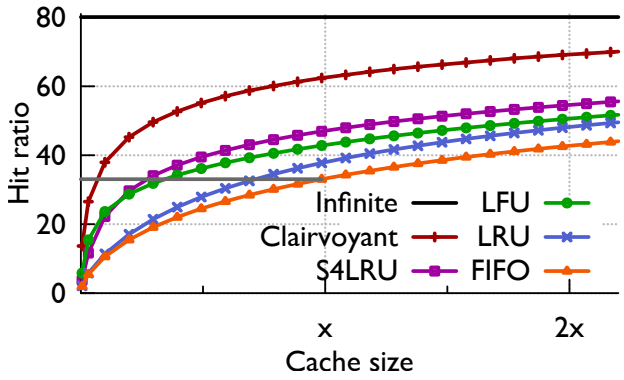


Figure 3.13: Simulation of Origin Cache with different cache algorithms and sizes.

ratio from going collaborative is 17.0% for FIFO and 16.6% for S4LRU. Compared to the current individual Edge Caches running FIFO, a collaborative Edge Cache running S4LRU would improve the byte-hit ratio by 21.9%, which translates to a 42.0% decrease in Origin-to-Edge bandwidth.

3.7.3 Origin Cache

We used our trace of requests to the Origin Cache to perform a *what-if* analysis for different cache algorithms and sizes. We again evaluated the cache algorithms in Table 3.4. The results are shown in Figure 3.13. The observed hit ratio for the Origin Cache is shown with a gray line and our estimated cache size for it is denoted size x.

The current hit ratio relative to the Clairvoyant algorithm hit ratio is much lower at the Origin Cache than at the Edge Caches and thus provides a greater opportunity for improvement. Moving from the FIFO cache replacement algorithm to LRU improves the hit ratio by 4.7%, LFU improves it by 9.8%, and S4LRU improves it by 13.9%. While there is a considerable 15.5% gap between S4LRU and the theoretically-almost-optimal Clairvoyant algorithm, S4LRU still provides significant traffic sheltering: it

reduces downstream requests, and thus Backend disk-IO operations, by 20.7%.

Increasing cache size also has a notable effect. Doubling cache size improves the hit ratio by 9.5% for the FIFO algorithm and 8.5% for the S4LRU algorithm. A double-sized S4LRU Origin Cache would increase the hit ratio to 54.4%, decreasing Backend requests by 31.9% compared to a current-sized FIFO Origin Cache. This would represent a significant improvement in the sheltering effectiveness of Facebook's Origin Cache. Combining the analysis of different cache sizes and algorithms, we see an inflection point in the graph well below the current cache size: the current hit ratio can be achieved with a much smaller cache and higher-performing algorithms. The current hit ratio (33.0%, in the portion of the trace used for simulation) can be achieved with a $0.7x$ size LRU cache, a $0.35x$ size LFU cache, or a $0.28x$ size S4LRU cache.

We omit the byte-hit ratio for the Origin Cache, but the difference is similar to what we see at the Edge Caches. The byte-hit ratio is slightly lower than the object-hit ratio, but the simulation results all appear similar with the exception of LFU. When examined under the lens of byte-hit ratio LFU loses its edge over LRU and performs closer to FIFO. The S4LRU algorithm is again the best for byte-hit rate with a 8.8% improvement over the FIFO algorithm, which results in 11.5% less Backend-to-Origin bandwidth consumption.

CHAPTER 4

EFFICIENT AND ADVANCED STATIC-CONTENT CACHING ON FLASH

4.1 Introduction

Although our study of Facebook's photo-serving stack [44] showed that advanced caching algorithms such as Segmented-LRU could significantly improve the cache hit ratios for Facebook's photo workload, such algorithms have not been implemented in production systems. The main reasons are that the cache replacement policies of these algorithms generate many small random writes, and that current NAND flash devices perform poorly with a large number of small random writes [67]. Today's flash devices typically use a large number of NAND flash chips in parallel to achieve high bandwidth. These parallel chips make the effective size of an erase block quite large, in the range of tens to hundreds of megabytes. Flash Translation Layer (FTL) typically reserves a large percentage of the storage capacity for such large erased blocks in order to reduce the waiting time for erasing a block before performing an actual write operation. When there are a large number of write requests, FTL's garbage collector struggles to keep pace, causing write amplification and long delays. For these reasons, the production system of the Facebook photo-serving stack employs a FIFO caching algorithm whose replacement policy generates sequential writes.

The crucial problem we would like to solve is to design a flash cache using advanced caching algorithms to achieve high caching hit ratios, without paying the penalty for heavily over-provisioning and triggering write amplifications. Ideally, we would like to solve this problem without special flash devices or special FTLs.

This chapter presents the design and implementation of a novel abstraction called

Restricted Insertion Priority Queue (RIPQ) that approximates a priority queue. Previous work showed that priority queue is a convenient abstraction for implementing several advanced caching algorithms [22, 87]. RIPQ's key mechanics include limiting insertion points, lazily moving items, and co-locating items with similar priorities. Limiting insertion points allows RIPQ to aggregate writes to insertion points in memory buffers until they are large enough to be efficient. Lazily moving items with updated priorities avoids fragmentation and excess writes. Co-locating items with similar priorities enables RIPQ to erase and overwrite large contiguous blocks. These features allow RIPQ to approximate the priority queue abstraction with mostly consolidated large writes to flash.

As a consequence of consolidating large writes, RIPQ requires moderate memory consumption. For example, a RIPQ with 20 insertion points and 256MiB buffer for each insertion point require ~5GiB for a 670GiB flash device. Such memory requirement could still be undesirable for a memory constrained environment. Therefore, we also present the design and implementation of a simplified abstraction called Single Insertion Priority Queue (SIPQ), which approximates RIPQ by using only one insertion point. In other words, SIPQ performs sequential writes to flash, and as a result requires minimal buffering. SIPQ maintains a separate priority queue in memory and provides an explicit trade-off between algorithm fidelity and write amplification via a logical occupancy parameter. SIPQ's single insertion point at the head of queue works well for algorithms such as LRU and its variations, but it does not work well for algorithms such as Greedy-Dual-Size-Frequency [26].

To evaluate RIPQ and SIPQ, we implemented the Segmented LRU algorithm [54] and Greedy-Dual-Size-Frequency (GDSF) algorithm. Facebook Origin Cache production system currently implements FIFO caching algorithm due to concerns regarding the

poor performance of small random writes on flash devices. However, our evaluations using the Facebook photo workloads show that RIPQ and SIPQ allow both algorithms to achieve substantially higher hit ratios while still performing efficiently on flash devices. Specifically, in the case of Facebook Origin Cache, hit ratios provided by Segmented-LRU algorithms with SIPQ improve by 7-8%, and GDFS algorithms with RIPQ also improve by 16.7-20%. Such improvements translate to 22.5-30% IOPS reduction to the backend storage tier.

RIPQ efficiently and faithfully implements these two algorithms on flash with 90% utilization of the device, incurs 1.2X write amplification, and achieves over 12K req/sec throughput. As a baseline, we use RocksDB, an open-source flash-based key-value system to store the cached data. RocksDB incurs over 5X write amplification and only achieves 2.9K req/sec throughput.

In addition, RIPQ-based Segmented-LRU would enable a traffic reduction of over 10% between Edge Cache and Origin Cache with less than 1.2X write amplification and over 14K req/sec throughput as well.

This chapter makes several contributions:

- The design and implementation of RIPQ, a framework for implementing advanced caching algorithms on flash without generating many small random writes.
- The design and implementation of SIPQ, a simplified RIPQ framework for implementing algorithms such as LRU in memory constrained environments.
- An evaluation on Facebook photo-cache traces that demonstrates advanced caching algorithms on RIPQ and SIPQ achieve excellent hit ratios, require small over-provisioning, and have small write amplification.

This chapter continues with background and motivation in Section 2.2. After con-

ducting a flash performance study in Section 4.2, we demonstrate the design of RIPQ in Section 4.3 and SIPQ in Section 4.4. We present an evaluation of our implementations in Section 4.5.

4.2 Flash Performance Study

Modern flash devices provide a unique combination of high capacity and high random-read throughput unseen in previous hardware, as well as a new set of limitations including block erasure [9] and wearing lifespan. Flash Translation Layer (FTL) is a firmware designed to perform mapping between the virtual space address provided by the flash and the physical address on the flash chips. Because the whole erase block in the flash memory has to be erased before new data can be written to it, the FTL often has to copy the valid data out before erasing and reusing one block, and thus write more data to the physical device than issued by the host. The ratio between the actual amount of data written by the FTL to the write issued by the host is called *FTL write amplification* [43]. Later we will also introduce the *implementation write amplification*, which is the ratio between the data written to the flash and data missed by the cache. The multiplication of FTL and implementation write amplification constitutes the overall system write amplification.

To explore the performance tradeoffs of flash, we conduct a performance study on the three types of flash cards listed in Table 4.1. We find that only sequential writes or large random writes can achieve sustained high throughput with high space utilization, regardless of the read/write ratio. The results inspire the design of RIPQ to issue only large writes, and SIPQ to issue only sequential writes.

Device	Capacity	Interface	Write (MiB/s)		Read (MiB/s)	Max perf
			Seq	Rand	Seq/Rand	write size (MiB)
FusionIO ioDrive	670GiB	PCI-E	589	76	794	512
Intel 320 Series	149GiB	SATA	161	18.8	264	256
LSI Warp 6208	1.86TiB	PCI-E	972	139	1514	512

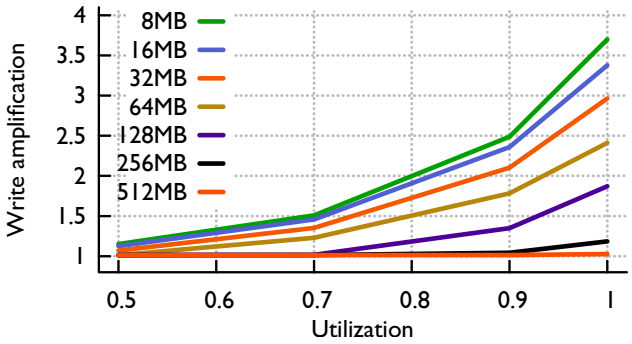
Table 4.1: Flash performance summary. Read/write are all 128KiB. Write results are the stable throughput after writing data 4 times the capacity of the device. Max-Throughput Write Size is the smallest write size (in the power of 2 series) required to achieve sustained maximum throughput at maximum capacity.

4.2.1 Random Write Experiments

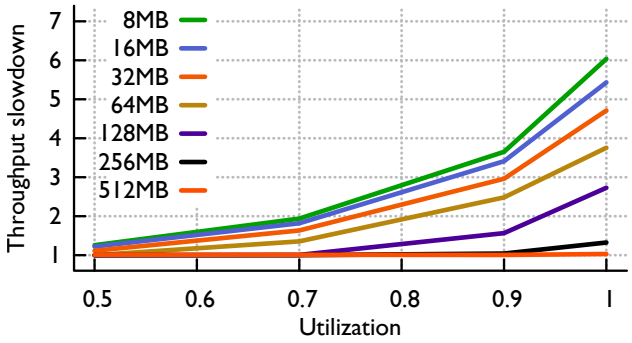
Direct caching algorithm implementations can yield random-write heavy workloads. To inform our design we conducted random write experiments that helped us understand the tradeoffs between random write size, space utilization, and FTL write amplification/throughput. In these experiments we performed aligned random writes of different sizes to the device under varying space utilization.

Figure 4.1a and Figure 4.1b show the results of FTL write amplification and throughput slowdown—i.e., the ratio of peak write throughput to the stabilized write throughput—for the random write experiments conducted on the FusionIO flash drive. The figures illustrate that as writes become smaller or space utilization becomes higher, throughput decreases and FTL write amplification increases. The striking similarity of curves suggests the decreasing throughput is highly correlated with the FTL write amplification.

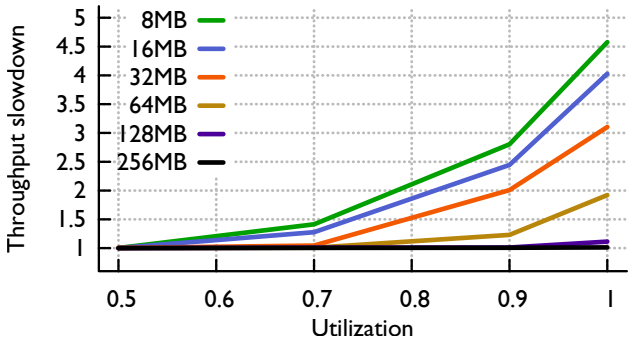
For Facebook photo cache, which has relatively low hit ratio and therefore is write-heavy, a decrease in write throughput can directly throttle the performance of the overall caching system. Moreover, high FTL write amplification reduces the lifespan of flash devices and with the continuing decrease in erasure cycles of large capacity flash cards [13, 29] the effects of FTL write amplification has worsened over time.



(a) Write amplification for Fusion



(b) Throughput slowdown for Fusion



(c) Throughput slowdown for Intel

Figure 4.1: Random write experiment on FusionIO and Intel flash.

We also obtained similar throughput slowdown result on the Intel device as in Figure 4.1c, however Intel doesn't provide the physical writes to the device so FTL write amplification is not available.

Min et al. [67] observed similar trends for three older devices where the random

write size had to reach 16MB or 32MB for the random write performance to match that of sequential writes. In our experiments this inflection point has continued to grow. Modern flash hardware consists of many parallel NAND flash chips [9] to improve throughput, and as a result aggregate erase block size on all the parallel chips can add up to tens or hundreds of megabytes, explaining the large write size needed to achieve high throughput and low FTL write amplification under high utilization of the device. This observation motivated us to design RIPQ to issue only large writes to SSD.

4.2.2 Sequential Write Experiment

A commonly used method to achieve sustained high write throughput on flash is by issuing only sequential writes. The FTL can effectively aggregate sequential writes to the parallel erase blocks [61], so if deletes or overwrites are on the parallel blocks, they can be erased without writing back any still-valid data resulting in low or no FTL-write-amplification. To confirm this we performed sequential write experiments to the flash devices. We observed sustained high performance for all write sizes above 128KB as reported in Table 4.1.[1] This combination of high write throughput, low FTL write amplification, and high space utilization inspires the design of SIPQ, which issues only sequential writes.

4.3 RIPQ

This section describes the design and implementation of Restricted Insertion Priority Queue (RIPQ). More specifically, we show how RIPQ approximates the priority queue

[1] FTL-write amplification is still low for smaller sequential writes, but they achieve lower throughput because they are bounded by IOPS instead of device bandwidth.

abstraction for flash hardware, present RIPQ's implementation details, and demonstrate that RIPQ framework efficiently supports advanced caching algorithms.

4.3.1 Priority Queue Abstraction

Priority queue has been identified by previous studies [22, 87] as a general and practical abstraction with a natural interface to implement various kinds of advanced caching algorithms. When priority queue abstraction is used to implement a cache, the priority queue holds all data items. The priority queue provides three operations for implementing caching algorithms:

- Insert(x, p): Insert a new item x with priority value p.
- update(x, p): Update the priority value of item x to p.
- delete-min(): Remove the item with the minimum priority value.

The priority of an item in the queue represents the utility of keeping the item in the cache. On a hit, *update* is called to adjust the utility of the accessed item. RIPQ creates a constraint that the priority of an item can only be *increased*. As we show later this is a mild constraint for the caching algorithms we consider. On a miss, *inserted* is called to add the item to the queue. If the insertion triggers a replacement, *delete-min* is called implicitly to evict an item with the minimum priority value from the cache.

To implement a specific caching algorithm with the priority queue framework, the implementer's job is to figure out how to define the utility of keeping a data item as priority. For example, if a caching algorithm can set the priority of an item based on its recently accessed time, accessed frequency, a combination of the two, or such combination with data object size. If a caching algorithm aims to optimize miss cost, then high

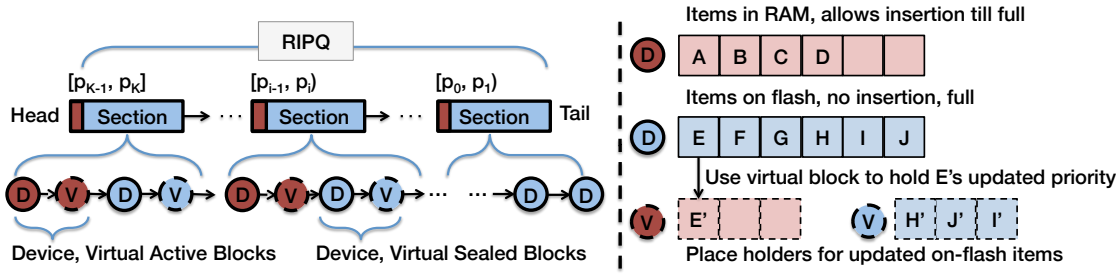


Figure 4.2: Overall structure of RIPQ.

priorities will be given to items with high miss cost. Greedy-Dual-Size [22] is a caching algorithm that sets a priority based on the likelihood of future access and object size.

However, for many caching algorithms, a straightforward implementation of a priority queue will generate many random small writes. It is not suitable for implementing a cache using flash devices.

4.3.2 Design of RIPQ

The main goal of RIPQ is to provide a framework to implement priority queue abstraction with mostly large writes, such that implementation on current flash devices will achieve high throughput with small over-provisioning.

RIPQ is a three-level structure as shown in Figure 4.2: queue, sections and blocks, where sections define the insertion points and a block is the unit of data written to flash hardware.

The highest level is a *queue* composed of K *sections*, with an insertion point at the head of each section. This design restricts where data can be inserted, so RIPQ only **approximates** the priority queue abstraction. The insertion error is bounded by $O(1/K)$. Since each insertion point uses memory to buffer exactly one block, the memory re-

quirement for buffering is KB, where B is the size of a block.

As shown in Figure 4.2, RIPQ splits the priority value range into K sections $[p_0, p_1), [p_1, p_2), \ldots, [p_{K-1}, p_K]$ where $p_0 = 0$ and $p_K = 1$. RIPQ natively supports a *relative priority queue* interface, where the priority is in $[0, 1]$ denoting the relative rank of an item in the queue. Support for *absolute priorities* will be introduced later. When an item is inserted into the queue, it is placed in the section whose range contains its priority value. For example, in a queue with sections corresponding to $[0, 0.3)$, $[0.3, 0.7)$ and $[0.7, 1.0]$, an item with priority value 0.5 would be inserted to the second section.

Each section consists of multiple blocks. A block is the basic unit for storing data items. There are two kind of blocks: device blocks and virtual blocks. A *device block* corresponds to a contiguous space on the flash devices, although the FTL on a flash device typically has another mapping to the physical location of flash hardware.

A *virtual block* is an in-memory place-holder of the new location of an item after its priority is updated. With virtual blocks, a priority update *virtually inserts* the item into the virtual block at the head of the appropriate section; it does not incur physical movements of data. We call this **lazy updates** because virtually inserted items are *reinserted* later when the device block of the item is evicted. Such reinsertions minimize the *write amplification* of RIPQ. Our experiments in a later section show that RIPQ creates small write amplifications even for complex caching algorithms.

A block is in either an active or sealed state. An *active device block* accepts writes (insertions) to an insertion point in the queue. Active device blocks buffer writes in memory until they are full, at which point they and their corresponding active virtual block transitions into the sealed state. A *sealed block* does not accept writes.

As shown in Figure 4.2, each section includes one active device block, one active

virtual block, and a list of sealed blocks. The active device block accepts new items being physically inserted into the section, and the active virtual block accepts virtual insertions. Each active device block requires a memory buffer of the same size until they are full, sealed and flushed to flash.

RIPQ provides a framework to achieve the desired tradeoffs of cache hit ratios, memory consumptions, and utilization and throughput of flash. The key parameter is K, which defines the granularity of insertions to the priority queue abstraction, and the amount of memory for buffering blocks. The large size of each block ensures high throughput and high utilization (or low over-provisioning) of flash.

4.3.3 Implementing Caching Algorithms

To demonstrate the flexibility of RIPQ, we implemented two families of advanced caching algorithms for evaluation: Segmented LRU [54], and Greedy-Dual-Size-Frequency [26], both of which yield major caching performance improvement for Facebook's photo workload. A summary of the implementation is show in Table 4.2.

Algorithm	**Interface Used**	**On Miss**	**On Hit**
Segmented-L LRU	Relative Priority Queue	`insert`$(x, \frac{1}{L})$	`update`$(x, \frac{\min(1,(1+\lceil p \cdot L \rceil)}{L}))$
Greedy-Dual-Size-Frequency L	Absolute Priority Queue	`insert`$(x, \text{Lowest} + \frac{c(x)}{s(x)})$	`update`$(x, \text{Lowest} + \frac{\min(L, n(x)) \cdot c(x)}{s(x)})$

Table 4.2: Segmented-LRU and Greedy-Dual-Size-Frequency with the priority queue interface provided by RIPQ.

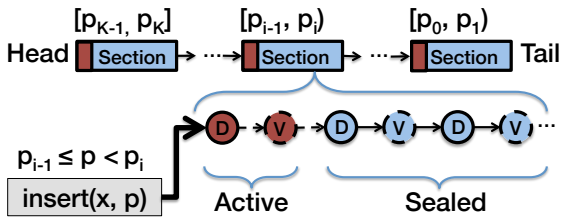
Segmented LRU Segmented-L LRU maintains L LRU caches of equal size. On a miss, an item is inserted to the head of the 1st LRU cache. On a hit, an item is promoted to the head of the next LRU cache, i.e., if it is in sub-cache l, it will be promoted to the

head of the $\min(l+1, L)$-th LRU cache. This algorithm demonstrated significant cache hit ratio improvements for the Facebook's Edge and Origin caches [44].

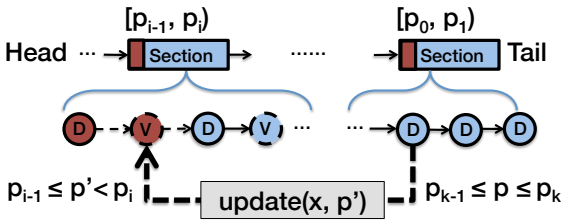
Implementing this family of caching algorithms is straightforward with the relative priority queue interface of RIPQ. On a miss, the missed item is inserted with priority value $\frac{1}{L}$. On a hit, RIPQ finds the previous priority of the accessed item, p, and updates it to $\min(1, \frac{1+\lceil p \cdot L \rceil}{L})$.

Greedy-Dual-Size-Frequency The Greedy-Dual-Size algorithm [22] provides a principled way to trade-off increased object-wise hit ratio with decreased byte-wise hit ratio by favoring smaller items. Such a trade-off is favored for the Origin cache within Facebook's photo-serving stack since the main purpose of Origin is to protect backend storage from excessive IO requests (Section 2.2). The Greedy-Dual-Size-Frequency [26] (GDSF) improves GDS by taking frequency into consideration. We use a variant of GDSF that caps the maximum value of the frequency of an item to L that performs better for our motivating workloads. The update rule of the algorithm is $p(x) \leftarrow \texttt{Lowest} + \frac{\min(L, n(x)) \cdot c(x)}{s(x)}$, where $n(x)$ is the number of accesses of an item since it was inserted to the cache. $c(x)$ is the cost of missing x. Because we are maximizing object-wise hit ratio we set $c(x) = 1$ for all items. GDSF uses the absolute priority queue interface of RIPQ.

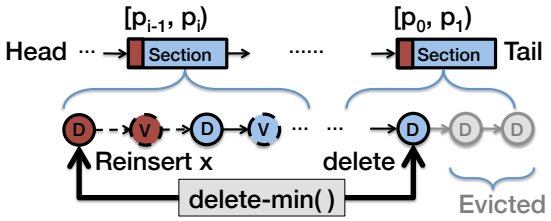
Limitations There are a few notable exceptions that are not implementable with a single RIPQ, e.g., MQ [89] and ARC [66], but they can be implemented with several RIPQs coexisting on the same hardware. A more problematic limitation comes from the update interface, which only allows increasing priority values. Algorithms that demote the priority value of an item on its access, such as MRU [27], cannot be implemented with RIPQ. MRU was designed to cope with scans over large data sets, which is irrele-



(a) Insertion



(b) Update



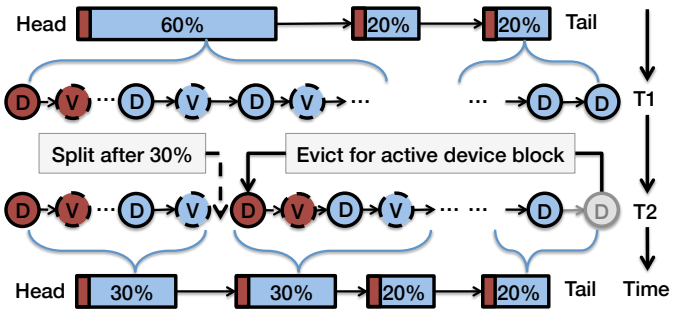
(c) Delete-min

Figure 4.3: Insertion, update, and delete-min operations in RIPQ.

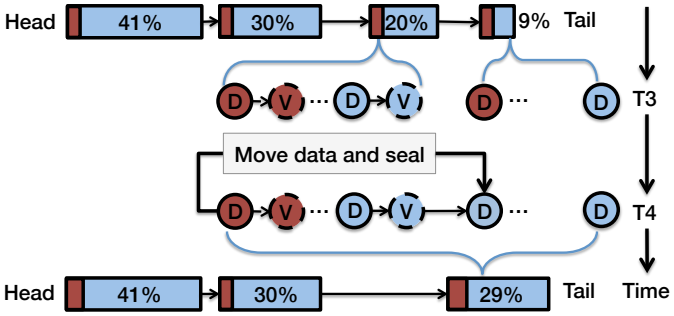
vant in the case of our research. RIPQ currently doesn't support delete operation, which is needed for a general-purpose read-write cache. But this limitation does not affect our motivating scenario within the static-content serving stack.

4.3.4 Implementation of Basic Operations

RIPQ implements the same three operations as a regular priority queue, using the data structures above.



(a) Section split process



(b) Section merge process

Figure 4.4: RIPQ internal operations.

Insert(x, p) RIPQ inserts the item to the active device block of section k that contains p, i.e., $p_{k-1} \leq p < p_k$. The write will be buffered until that active block is sealed. Figure 4.3a shows an insertion.

Update(x, p) RIPQ avoids moving item x that is already resident in a block in the cache. Instead, RIPQ virtually inserts x into the active virtual block of section k that contains p, i.e., $p_{k-1} \leq p < p_k$[2]. The virtual item in section k holds the updated location of item x. RIPQ also removes x from its previous section j: if it is a virtual item in j, we remove it; if it is a physical item, we mark it as moved. Figure 4.3b shows an update.

One constraint of RIPQ is that an item's priority can only be increased, not decreased. The physical block that contains an item is evicted when it has the lowest

[2]The exception is when $k = K$, $p_{k-1} \leq p \leq p_k = 1$.

priority in the queue. Thus, RIPQ's lazy update can only move items to higher priorities. This constraint is natural for many caching applications, especially Web caches, because most workloads exhibit time locality, i.e., access of an item indicates it is more likely to be accessed again [50]. Most caching algorithms exploit this workload characteristic, with the notable exception of the Most-Recently-Used (MRU) cache algorithm. MRU is designed to handle scan behavior, which does not happen in our motivating workloads.

`Delete-min`() This operation takes place implicitly in RIPQ when an active block is full and flushed to the flash. The lowest-priority block in queue is evicted to create space and is physically overwritten by the flushed block. Before overwriting the evicted block RIPQ *materializes* any items in it that were virtually inserted to higher positions in the queue by physically reinserting them to the sections that contain their virtual blocks and deleting their virtual items. Figure 4.3c shows a delete-min.

These reinsertions help preserve caching algorithm fidelity, but cause additional writes to flash, noted earlier as the implementation write amplification earlier. RIPQ can explicitly trade lower caching algorithm fidelity for lower write amplification by not reinserting items whose priority is smaller than a given threshold, i.e., in the last 5% of the queue.

Internal operations RIPQ internally maintains invariants to ensure bounds on memory consumption and approximation error. In particular, RIPQ controls (1) the number of sections, and thus active device blocks and (2) the size of each section. Each active device block has a partially full write buffer that is consuming RAM. The size of each section is directly proportional to the approximation error of RIPQ.

To maintain both invariants, RIPQ splits and merges adjacent sections as shown in Figure 4.4. The α parameter controls the average size of sections, where $0 < \alpha < 1$. RIPQ splits a section when its relative size has reached 2α. For example, if $\alpha = 0.3$ then a section from $[0.4, 1.0]$ would be split into two sections of $[0.4, 0.7)$ and $[0.7, 1.0]$ respectively, as shown in Figure 4.4a. RIPQ merges two consecutive sections if the sum of their sizes is smaller than α (show in Figure 4.4b). The *relative size* of a section is a ratio measured based on the number of items or the total byte size of items in it. These operations guarantee that (1) there are at most $\lceil 2/\alpha \rceil$ sections, and (2) each section is no larger than α.

No data is moved on flash for a split or merge. Splitting a section creates a new active device block with a write buffer and a new active virtual block. Merging two sections combines their two active device blocks: the write buffer of one is copied into the write buffer of the other. Splitting happens often and is how new sections are added to queue as sections at the tail are evicted block-by-block. Merging will be rare because it requires the total size of two consecutive sections to shrink from 2α (α is the size of a new section after a split) to α to trigger a merge and the amortized cost of merge per operation is only $O(1/(\alpha M))$.

4.3.5 Other Implementation Details

Supporting Absolute Priorities RIPQ naturally supports the LRU family of algorithms such as LRU, Segmented LRU [54], because they only require a few relative priority insertion points. But caching algorithms such as LFU, SIZE [8], and Greedy-Dual-Size[22] require the use of absolute priority values when performing insert and update. RIPQ supports absolute priority values with a mapping data structure to trans-

late absolute values to relative priority values. The data structure maintains a dynamic histogram that returns approximate percentiles. The percentile values are used as the internal priority values. The histogram consists of a set of bins and merges/splits bins dynamically based on their relative sizes, similar to the way we split/merge sections in RIPQ.

In-memory and flash data layout RIPQ uses in-memory data structures to track the ordering of the queue, the size information and positions of blocks and sections, and to buffer active blocks. Here the size information includes both the number of items and the total byte size of those items in that block/section. The ordering of the queue in RIPQ's three-layer hierarchy is represented with two levels of ordered sets in RAM: the queue is an ordered set of sections, and each section is an ordered set of blocks.

RIPQ stores the size information and physical locations of sealed device blocks in memory. Active device blocks are buffered in memory until they are full and then transition to sealed device blocks with physical locations. Virtual blocks do not have physical locations, so RIPQ only tracks their size information. RIPQ also tracks the size of sections so it can calculate each section's priority range.

The items in sealed device blocks are stored on flash, each of which corresponds to a contiguous space in the flash logical address space. During an eviction RIPQ locates the device block with the lowest priority through its in-memory representation. It then reads the on-flash block header to get the *ids* and *offsets* of all items in that block. RIPQ decides whether to reinsert items by querying another in-memory structure: the item index. Once items are reinserted (to in-memory write buffers) the evicted block is entirely overwritten by a full active device block that is flushed to the flash.

Parameter	Description and Goal	Our Value
Block Size (B)	To satisfy sustained high random write throughput.	256MiB
Number of Blocks (M)	Flash caching capacity divided by the block size.	2400
Average Section Size (α)	To bound the number of sections $\leq \lceil 2/\alpha \rceil$ and the size of each section $\leq 2\alpha$, trade-off parameter for insertion accuracy and RAM buffer usage.	0.05
Insertion Points (K)	Same as the number of sections, controlled by α and proportional to RAM buffer usage.	20

Table 4.3: Key Parameters for RIPQ

Item index The *item index* is an in-memory hash table from item ids to metadata about each item in the cache. This includes the physical device block, offset, and length of an item to facilitate reading the item for hits in the cache. It also includes the virtual block number for items whose priorities have been updated to facilitate reinserting items into queue. A typical RIPQ implementation has at most thousands of blocks, so the virtual block number can be represented with 2 bytes. In contrast, an exact caching algorithm would use larger amounts of memory. For instance, LRU would need two pointers for each item stored in cache for the forward and backward pointers of each item in the LRU chain. Chapter 2 discusses the large body of related work on minimizing the memory consumption of indexing structures for flash or spinning disk [11, 12, 14, 63, 64, 73].

Parameters Table 4.3 lists the parameters of RIPQ, along with a description about the purpose of each and the value we picked for our implementation. The *block size B* is chosen to surpass the threshold for a sustained high write throughput for random writes, and the *number of blocks M* is calculated directly based on cache capacity. The number of blocks affects the memory consumption of RIPQ, but this is dominated by the size of the write buffers for active blocks and the indexing structure. These active blocks correspond directly to the number of *insertion points K* in the queue. The *average section size* α is used by the split and merge operations to bound the memory consumption and approximation error of RIPQ.

Durability The cache is intended as volatile storage for the Facebook photo serving stack, so durability is not a requirement but an added bonus. However, because the queue is materialized on flash, even after a power loss, all the data can still be recovered from the flash (except for those in the RAM buffer). The order information of blocks/sections can be periodically flushed to flash as well, so we can keep the priority information of physical items.

4.4 SIPQ

A potential drawback of RIPQ is the memory required for buffering multiple large blocks. We propose a simplified RIPQ abstraction called Single Insertion Priority Queue (SIPQ) for the memory-constraint environment. SIPQ is suitable for use with LRU family of algorithms, but not ideal for use with more complex algorithms like Greedy-Dual-Size-Frequency.

4.4.1 Design of SIPQ

A key principle for SIPQ is the use of sequential writes instead of large random writes to achieve high write throughput. This principle avoids the need for write buffers and incurs almost no FTL write amplification because it treats flash storage as a cyclic log-structured store [76].

However, the single insertion point also severely restricts how SIPQ arranges data on flash. To preserve a priority ordered queue structure, SIPQ maintains a separate *logical priority queue* in memory for tracking item priorities, while leaving the *device queue* unordered. The logical queue will only remember the key, length and priority of each

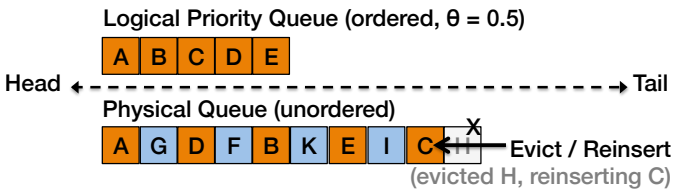


Figure 4.5: How SIPQ works.

item, while the actual data is stored in the device queue.

Separating the logical priority queue and unordered device queue causes implementation write amplification during eviction from the device queue: a potentially large portion of items needs to be reinserted to preserve the algorithm fidelity. SIPQ allows developers to explicitly trade-off algorithm fidelity for implementation write amplification through a parameter, *logical occupancy* or θ: the ratio between the total size of items in the logical priority queue and the total size of device queue. $\theta = 1$ would mean we try to keep all the items in the priority order, while $\theta = 0.5$ would mean half of the items on a device are not ordered and can be evicted without reinsertion. As shown in Section 4.5, simple caching algorithms such as LRU can be implemented on SIPQ with a low θ value (0.5) without losing much performance, while achieving a low implementation write amplification (e.g., 1.08).

4.4.2 Implementation

SIPQ implements the same set of operations as RIPQ.

`Insert`(x, p) Insert the key of x into the logical queue with its priority p. If the data is not on the device, we append the data of x to the head of the device queue.

Update(x, p) Only update the priority of of x in the logical queue. This operation does not incur any physical write to the flash hardware.

Delete-min() This operation occurs when SIPQ triggers a cache replacement and proceeds on items in both queues. If the logical queue exceeds its size limit, the item with the smallest priority will be evicted only from the logical queue. When the device queue is full, SIPQ evicts items from the tail of the queue to create space for new items. As SIPQ evicts items from the device queue it checks for their existence in the logical queue.

Figure 4.5 shows a simplified SIPQ structure with 0.5 logical occupancy (θ) value. The upper logical queue occupies 0.5 of the total space on flash, and only A, B, C, D, E are ordered by their priorities in the logical queue. In the figure, SIPQ is working at the tail of the device queue, evicting J, H while reinserting the active item C to the head of the device queue.

In memory data-structure In addition logical queue, SIPQ also uses a hash table to map from keys to metadata that includes the location of the data on flash and the item's state of being in device only or in both queues.

Durability All the data in the device queue can be recovered after a power loss for SIPQ, but because the logical queue is separated from the device queue we lose most of the priority information.

4.5 Evaluation

Our evaluation answers three key questions: (1) What is the impact of RIPQ and SIPQ's approximations of caching algorithms on hit ratios, i.e., what is the effect on algorithm fidelity? (2) What is the write amplification caused by RIPQ, SIPQ, and a LSM-baseline? (3) What is the throughput performance of RIPQ, SIPQ, and the LSM-baseline?

4.5.1 Experimental Setup

Hardware Environment Experiments are run on servers equipped with a FusionIO ioDrive 720GB flash device and 144GB DRAM space. All flash devices are configured with 90% space utilization, leaving the remaining 10% for the FTL.

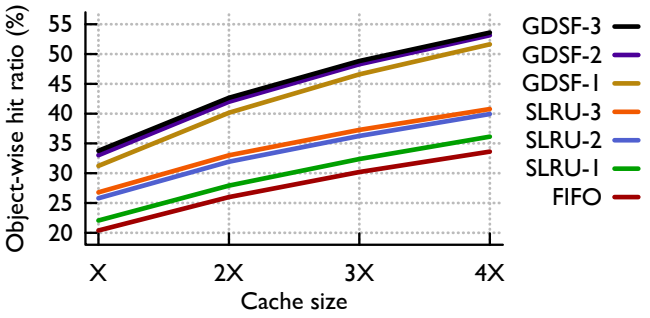
Baseline: RocksDB We compare RIPQ and SIPQ to a baseline solution: RocksDB [7]. RocksDB is a popular open source flash-based key-value store built on LevelDB [4] using Log-Structured Merge-Trees (LSM) [72], and can be used to implement caching algorithms as an object store. We believe the performance of RocksDB is indicative of the performance of LSM-solutions in general. It should be noted that our comparison is not entirely fair to RocksDB and the general LSM family solution because they are designed for more general workloads and support efficient range queries. Nevertheless, we felt it was still important to compare against a popular type of existing storage solution that is possible to be used in caching and has been optimized on flash devices.

Framework Parameters RIPQ uses a 256MiB block size to achieve the max write performance based on our performance study of ioDrive. It uses $\alpha = 0.05$, i.e., 20 sections, which provides a good trade-off between the fidelity to the implemented algorithms and the total RAM space RIPQ uses for buffering: 256 Mib x 20 = 5GiB (which is moderate for a typical server).

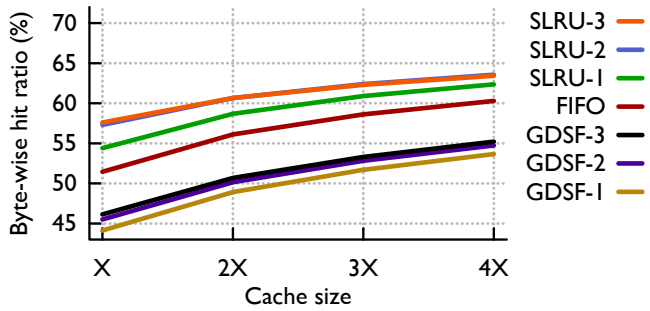
SIPQ also uses a 256MiB block size so the number of blocks on flash is the same as RIPQ. Because SIPQ issues only sequential writes, this buffer and the write size could be further shrunk without adverse effects. Two *logical occupancy* values are used in evaluation: 0.5, and 0.9, and each represents a different trade-off between approximation fidelity and implementation write amplification. Later, these two setting are noted as SIPQ-0.5 and SIPQ-0.9, respectively.

Caching Algorithms Two families of advanced caching algorithms are evaluated: Segmented-LRU (SLRU) [54] and Greedy-Dual-Size-Frequency (GDSF) [26]. For Segmented-LRU, we vary the number of segments from 1 to 3, and report their results as SLRU-1, SLRU-2, and SLRU-3, respectively. We similarly use 1 to 3 segments For Greedy-Dual-Size-Frequency, denoted as GDSF-1, GDSF-2, and GDSF-3. Description of these algorithms and their implementations on top of the priority queue interface are explained in Section 4.3.3.

Facebook Trace Two sets of 15-day sampled traces collected within the Facebook photo-serving stack are used for evaluation, one from the Origin Cache, and the other from a large Edge Cache facility. The Origin trace contains over 4 billion requests and 100TB worth of data, and the Edge trace contains over 600 million requests and 26TB worth of data. To emulate the effects of different cache capacities to a fixed-sized flash device, we further downsampled the trace by $1, 1/2, 1/3$, and $1/4$, to approximate a



(a) Object-wise hit ratios on Origin trace.



(b) Byte-wise hit ratios on Edge trace.

Figure 4.6: Exact algorithm hit ratios on Facebook trace.

cache size of $1X, 2X, 3X$, and $4X$ for later experiments. During both experiments and simulations, we use the first 10 days to warm up the cache and measure performance over the following 5 days.

4.5.2 Results of Facebook Trace

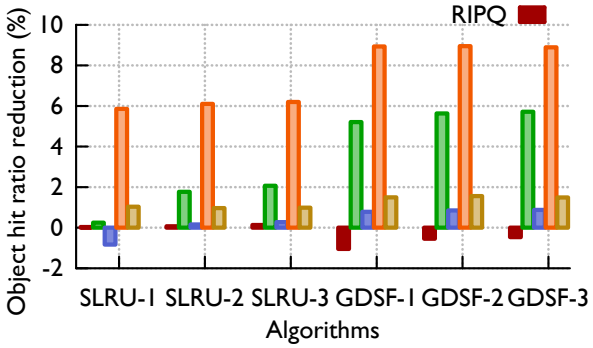
This section presents results from running real experiment with the Facebook trace to evaluate the algorithm fidelity, write amplification, and throughput of RIPQ, SIPQ, and RocksDB. Exact algorithm results are obtained via simulation. We report the hit ratio to optimize at each cache, i.e., object-wise hit ratio for the Origin trace and byte-wise hit ratio for the Edge trace.

Performance Baseline of Exact Algorithms We first investigate the hit ratios achieved by the exact caching algorithms to determine the baseline for our algorithm fidelity evaluation. Results are shown in Figure 4.6. FIFO, which is fully sequential and has no FTL write amplification, is currently deployed in the Facebook stack.

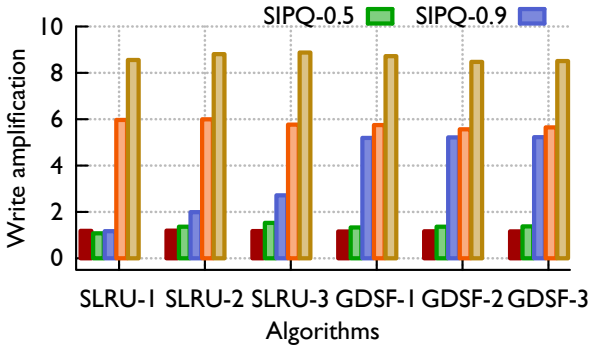
As the Facebook photo workload has changed since our previous study [44], the best algorithm to replace FIFO for the Origin and Edge is no longer the same. For the object-wise hit ratio on Origin trace, Figure 4.6a shows that Greedy-Dual-Size-Frequency family algorithms outperform Segmented-LRU and FIFO by a large margin. At $2X$ cache size, GDSF-3 improves the hit ratio over FIFO by 16.7%, which creates a reduction of IOPS to the backend by 22.5%. For the byte-wise hit ratio on Edge trace, Figure 4.6b shows that Segmented-LRU is still a better option. Again at $2X$ cache size, SLRU-2 improves the hit ratio over FIFO by 4.5%, which results in a bandwidth reduction between Edge and Origin by 10.2%. Greedy-Dual-Size-Frequency performs poorly on the byte-wise metric because it down-weights large sized photos.

Approximation Fidelity Figure 4.7a and 4.8a show the hit ratio reduction of different algorithms implemented on top of RIPQ, SIPQ-0.5, SIPQ-0.9, RocksDB-0.5, and RocksDB-0.9, compared to their exact performance baselines with the $2X$ cache size setup from Figure 4.6. In general, a small reduction indicates high approximation fidelity achieved by the underlying framework.

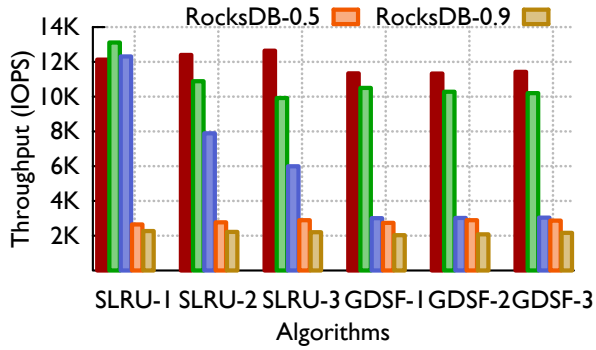
RIPQ consistently achieves high approximation fidelity for the SLRU family algorithms, and its hit ratio reduction values are below 0.2% for both object-wise metric on Origin trace and byte-wise metric on Edge trace. For the GDSF family, RIPQ's algorithm fidelity becomes lower because the algorithm's complexity increases. The greatest infidelity seen for RIPQ is a 5% difference on the Edge trace for GDSF-1. Interestingly,



(a) Object-wise hit ratio reduction



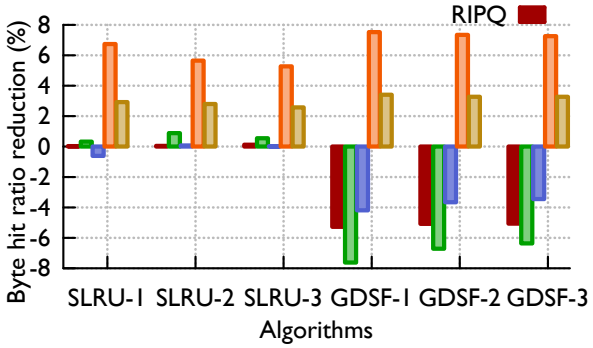
(b) Write amplification



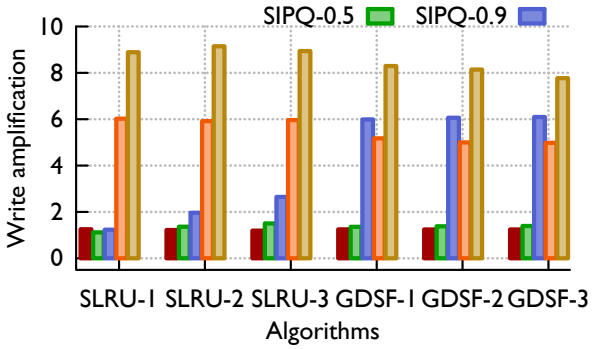
(c) IOPS throughput.

Figure 4.7: Performance of RIPQ, SIPQ, and RocksDB on Origin.

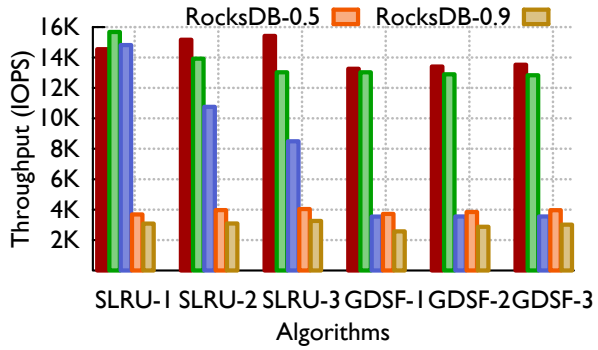
for the GDSF family of algorithms, the "infidelity" generated by RIPQ improves hit ratio. The large gain on byte-wise hit ratio is explained by the fact that the exact GDSF algorithm is designed to trade byte-wise hit ratio for object-wise hit ratio by favoring small items, and approximation improves the byte-wise hit ratio. RIPQ-based GDSF



(a) Byte-wise hit ratio reduction.



(b) Write amplification.



(c) IOPS throughput.

Figure 4.8: Performance of RIPQ, SIPQ, and RocksDB on Edge.

family also only incurs a 1% reduction in *object-wise* hit ratio. Overall, RIPQ has high algorithm fidelity.

SIPQ and RocksDB both have high fidelity when the occupancy/utilization parameter is set to 0.9, which means 90% of the caching capacity is managed by the exact

algorithm. RocksDB with low utilization parameter 0.5 loses much in terms of hit ratio, as its cache capacity is effectively reduced by a half. SIPQ-0.5, on the other hand, still achieves a decent fidelity for SLRU algorithms, only resulting in a 0.24% object-wise hit ratio reduction for SLRU-1 to 2.8% object-wise hit ratio reduction for SLRU-3 on Origin, and 0.3% byte-wise hit ratio reduction for SLRU-1 to 0.9% byte-wise hit ratio reduction for SLRU-3 on Edge. These algorithms tend to put new and recently accessed items towards the head of the queue, which is similar to the way SIPQ inserts and reinserts items at the head of the device queue. However, SIPQ-0.5 also shows large fidelity loss for the GDSF family, causing object-wise hit ratio decrease on Origin and byte-wise hit ratio increase on Edge. For these algorithms the items can have very different priority values because of their different sizes even if they enter the cache at the same time, and SIPQ's approach of inserting items in one place is a poor approximation.

Write Amplification Figure 4.7b further shows the combined write amplification (FTL write amplification multiplied by implementation write amplification) of different frameworks. RIPQ consistently achieves the lowest write amplification, with an exception for SLRU-1 where SIPQ-0.5 has the lowest value for both traces. This does not come as a surprise because SLRU-1 (LRU) is a simple algorithm and SIPQ-0.5 has low logical occupancy. The write amplification of RIPQ is largely stable regardless of the complexity of the caching algorithms, ranging from 1.18 to 1.25 for the SLRU family, and from 1.15 to 1.25 for the GDSF family, for both traces.

SIPQ-0.5 achieves moderately low write amplifications at the cost of fidelity reductions for complex algorithms. SIPQ-0.5's write amplification value also increases with the algorithm complexity. For SLRU, the write implementation for SIPQ-0.5 rises from 1.08 for SLRU-1 to 1.52 to SLRU-3 on Origin, from 1.11 for SLRU-1 to 1.50 to SLRU on Edge. For GDSF, the value ranges from 1.33 for GDSF-1 to 1.37 to GDSF-3

on Origin, and from 1.36 to 1.39 on Edge. Results for SIPQ-0.9 observe higher write amplification, especially for the GDSF family, but show a similar trend.

RocksDB has high write amplification even at a low utilization (0.5). The primary reason is that RocksDB's garbage collection algorithm is compacting new files regularly into a few Sorted String Tables to facilitate faster range queries, but this is unnecessary for our cache workloads. In this unfair comparison, RocksDB-0.5 has a write amplification between 5 and 6 for all algorithms, and the value for RocksDB-0.9 is between 8 and 9.

Cache Throughput Throughput results are shown in Figure 4.7c and 4.8c. RIPQ and SIPQ-0.5 consistently achieve over 10 000 requests per second on Origin, but SIPQ-0.9 fails to do so for the GDSF family, and RocksDB has the lowest throughput in all cases. This performance is highly related to the write amplification results because in all three frameworks (1) workloads are write-heavy with below 63% hit ratios, and (2) write amplification proportionally limits the write throughput, which further throttles the overall throughput.

CHAPTER 5

CHARACTERIZING LOAD IMBALANCE IN GRAPH CACHE

5.1 Introduction

In-memory caches are often used to scale modern Web services since they can decrease request latency for users and relieve load on storage and database servers. Instead of executing a potentially resource-intensive operation on the storage or database back-end directly, clients first consult an appropriate cache server for a copy of the desired data.

The aggregate request volume at popular websites would significantly overwhelm the capacity of a single cache server. As such, data is normally divided among hundreds or thousands of cache servers [20, 34], typically by partitioning the large space of possible object IDs into segments that are then mapped onto cache servers. The segments — called *shards* — commonly contain a large number of objects to reduce the size of the object-to-server lookup map. Further, "related" objects (*i.e.*, benefiting from co-location on the same server) tend to be mapped to the same shard to mitigate the impact of *thundering herds* [69] and decrease query fan-out [20].

Ideally, cache servers would all observe similar request rates (volume per unit time), since this would provide predictable request latencies [42] and reduce the over-provisioning of resources necessary to withstand peak workloads [90, 46]. In reality, however, segments confronted with real-world workloads often sustain variable and dynamic request rates that can contribute to significant load imbalance. Imbalance can be caused by the skewed access popularity among different objects (*e.g.*, Facebook's social-graph objects reveal varying access popularity), as well as the decision of co-locating related data within the same segment to support more advanced data queries (*e.g.*, each shard

in Facebook's cache-tier co-locates related objects for more efficient range queries and cache consistency maintenance).

There are many options for balancing the number of objects apportioned to servers in distributed caches. Most mechanisms randomly partition data across servers by hashing [55, 56], while some additionally adapt to changes [32, 46]. Much recent interest has also focused on understanding and mitigating hot spots and load imbalance that arise in skewed workloads seen in key-value stores and cache systems [34, 42], but to the best of our knowledge no comprehensive analysis to date permits online analysis of the key culprits based on a real workload.

In this chapter, we investigate the nature of load imbalance based on a real-world setting at Facebook's social-graph cache, TAO, where data have skewed access popularity and cannot be randomly separated. We explore how different categories of approaches — fine-tuned consistent hashing and hot content replication (including a special case for front-end caching) — might help to mitigate their impact and investigate the limitation of each approach category.

This chapter offers the following contributions:

- We determine that the popularity skewness at object level is not a major cause of cache load imbalance.
- We deduce that load imbalance in systems like TAO can stem from a combination of load-insensitive partitioning, extremely hot shards, and random temporal effects.
- We survey current approaches to load balancing and assess their effectiveness on our real-world traces through simulation to guide future research.

5.2 Analyzing Load Imbalance

A good understanding of load-imbalance situations should lead to a proper justification of the its root causes as well as better reasoning about the solution space. To accomplish this, we use in-memory cache-access traces from Facebook's graph-storage, TAO.

5.2.1 Environment

To provide general background, Figure 5.1 illustrates a typical architecture of in-memory caching facilitated modern Web stack. When a front-end Web server receives Web clients' HTTP requests, it often spawns numerous data fetching requests to generate a content-rich response. To reduce the fetching latency and database querying overhead, such data fetches are first directed to a tier of caches, where popular content results reside in DRAM. Only if the requested data does not reside in the caching space would a data request reach the backend. Based on the manner in which the backend request is redirected, there are two categories of caching tiers: read-through and read-aside. The request flow in Figure 5.1 follows the read-through style, within which the cache serves as the proxy to fetch data from the backend while caching is on the way. In the read-aside style, Web server takes the responsibility to request the data from backend servers and fill the content in cache later. TAO, the graph storage solution at Facebook organizes its cache tier as read-through caches, and specifically has two layers of caches: follower and leader. The follower cluster co-locates with each Web front-end cluster, while a leader tier cluster serves an entire region's followers.

Sharding. Sharding or partitioning is a general approach to scale a database beyond any single server's capacity and is commonly used in production caching systems

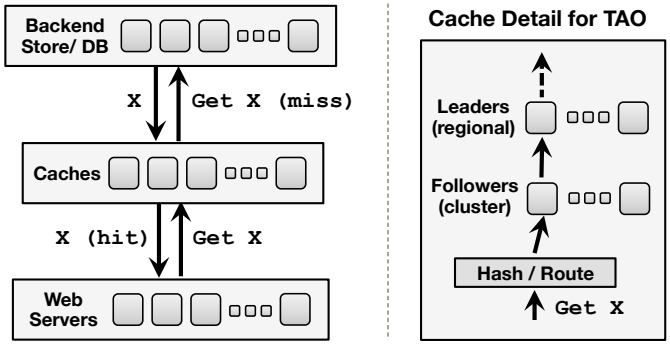


Figure 5.1: Cache architecture. Left shows how read-through cache serves between the front-end Web servers and the backend; right shows how Facebook's TAO cache is organized.

[69, 20]. There are two ways sharding can be operated depending on the choice of co-locating related data — *random sharding* and *dependent sharding*. In a normal key-value interface system, such as `memcached`, random sharding is sufficient for distributing relatively similar numbers of objects to each shard, and the independence of objects within the same shard does not impact its GET/SET operation performance. However, dependent sharding is more appealing for systems that provide advanced queries based on structural data. TAO, for instance, provides social graph oriented range queries which constitutes over 43.6% [20] of its operations. By using dependent sharding to co-locate socially connected graph objects, TAO is able to reduce the range-query fan-out and therefore network overhead. Moreover, dependent sharding also provides the possibility of consistency tracking on related data. Our analysis results in this chapter are more relevant to the dependent sharding caches, where the access disparity between different shards may be more profound than the disparity of different cached items. Figure 5.2 shows an example of dependent sharding for graph. Ignoring the sharding type, shard-to-server mapping is often conducted through consistent hashing.

Traces. Throughout the rest of this chapter, we rely on three sets of TAO production traces collected between front-end Web servers and TAO followers. Table 5.1 gives a

Trace	Sampling Basis	Sampling Period	Source	Details
TopShards	Time	1 min	Cache	Traffic on top 100 shards and 20 objects
ReqSample	Request	100K reqs	Web Server	Requests sent to cache cluster
ServerTraffic	Time	4 min	Cache	Reported server load in req/sec

Table 5.1: Tao trace summary: TopShards reports hot contents traffic. ReqSample samples requests from Web servers. ServerTraffic contains cache load snapshots.

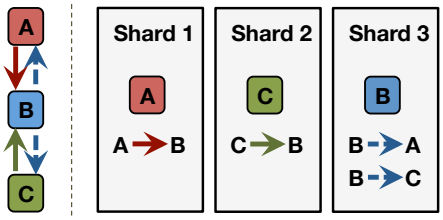


Figure 5.2: Dependent sharding on a directional graph: edges are co-located with their source vertex within the same shard.

demonstration of each, and their use in this study is explained below.

- TopShards is directly reported by each cache server, constituting the exact number of requests sent to the 100 most popular shards and 20 most popular objects. This trace gives us a high-quality source to analyze the impact of load imbalance due to popularity skewness and the temporal dynamics of popular objects/shards.
- ReqSample is sampled among the real read requests sent from Web servers. While the per-request sampling gives a less-detailed signal for a temporal analysis on popular objects/shards, the full fidelity in terms of the request flow provides a good basis for simulation.
- ServerTraffic is a per-cache traffic rate report collected every four minutes, as an adjunct to the TopShards. Thus ServerTraffic reports the entire cache traffic instead of hot contents only. It serves as ground truth in our study, validating the cache-load status from the TopShards-driven analysis and the ReqSample-driven simulation.

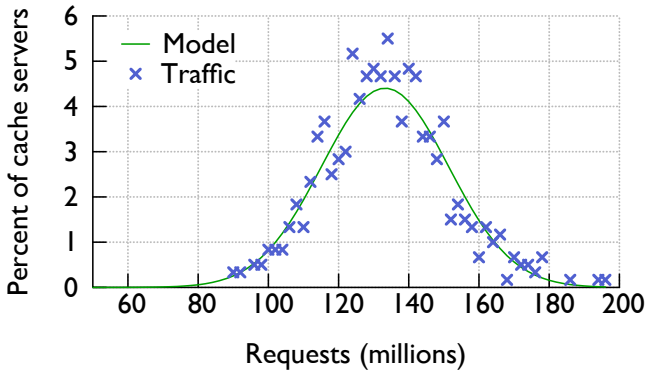


Figure 5.3: Load distribution (a cluster).

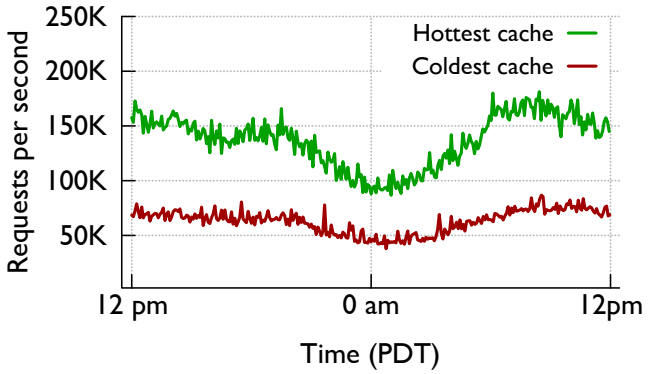


Figure 5.4: Load disparity within a day.

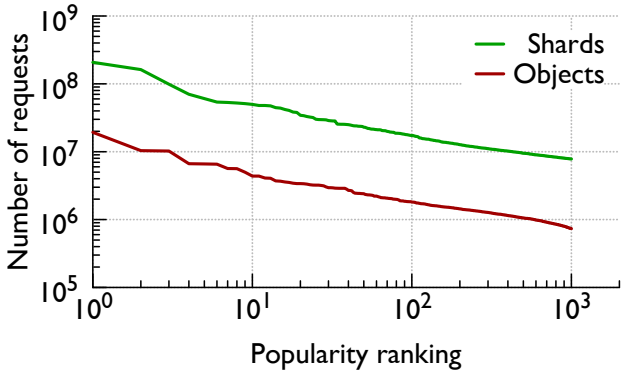


Figure 5.5: Content popularity.

5.2.2 Analysis

TAO already deploys several load-balancing techniques [20], however, it still experiences a certain degree of load imbalance within the caching tier; Figure 5.3 quantifies this,

within a follower cluster from TopShards, discussed earlier. Statistically, across the cluster we profiled, the request load over a 24-hour period is normally distributed with parameters $\mu = 1.342\,43 \times 10^8$ requests and $\sigma = 1.810\,87 \times 10^7$ ($p < 0.02$, Shapiro-Wilk normality test). Figure 5.4 further illustrates the traffic dynamics of both the hottest and coldest server from the same cluster. As can be seen, the disparity exists throughout the day, as both curves follow a similar diurnal pattern, though the contrast is greater at peak hours (>100K requests per second [rps] difference) and smaller during idle period (<50K rps difference).

In order to find the root cause (or causes), we examined traces to mainly answer the following two questions:

- Does skewed content popularity impact load disparity in the cache?
- Does imbalanced placement of similarly popular content play a major role?

Skewed content popularity. We used our traces to confirm that content popularity, defined in terms of number of accesses, resembles a power-law distribution across two different populations: **(1)** distinct objects in the social graph (vertex and edge); **(2)** different shards, each of which maintains some partition of the graph). Figure 5.5 shows the number of requests for each of the top-1000 objects and top-1000 shards, as reported by cache servers. In light of this, does the hottest object become a dominant resource bottleneck for a cache server? Figure 5.6 shows the traffic dynamics of the most popular object and its associated shard. We note that a single hot object can contribute almost the entire traffic for its hosting shard, sufficient to push the shard into the top-100. However, notice the level of imbalance: the traffic associated with this hot object wouldn't even be half the available capacity for the coldest cache server. The skewed popularity among different objects does not appear to be as a major cause of cache-load imbalance for the

request traffic between front-end Web servers and cache servers. As further discussed in Section 5.3, this is also why an additional layer of front-end cache has limited impact in balancing the cache load.

Figure 5.7 examines the traffic dynamics of the six top-ranked shards, based on their popularity. Compared to object-traffic (Figure 5.6), popular shards receive significantly higher load due to the combination of many related objects, and has much more impact on the cache server. It is noteworthy that each curve in Figure 5.7 combines multiple reports from servers that all hold replicas of the same shard.

Our question is made more complex because TAO itself has an architectural feature that comes to bear here. Within TAO, shard replication is such that a hot shard will be spread over multiple cache servers, and the front-end router redirects Web-server requests among them. Specifically, the hottest shard has been replicated 10-fold throughout the 24 hour period. Such replication starts when a server has less than 20% remaining CPU- and network- capacity, while more than 25% of the request load comes from a "dominant" shard. In other words, as long as a shard contributes 20% of a busy server's capacity cap it needs to be replicated. From our analysis, TAO replication plays a significant role in keeping servers' loads below 200K rps.

Some aspects of the existing replication mechanism need improvement: **(1)** if a shard becomes hot too quickly, replication is too slow to react, and **(2)** the current reverse routine (de-allocating server space for no-longer-hot shards) is too conservative. Figure 5.8 shows traffic dynamics for a "popularity-surged shard" and the involvement of up to five extra replicas to split the load for a 48-hour period. When the traffic to the popularity-surged shard rises from <75K rps to >175K rps within ten minutes, the replication process starts (but only after the original server stays at that load for another eight minutes); replication continues an hour later when the shard is still causing too

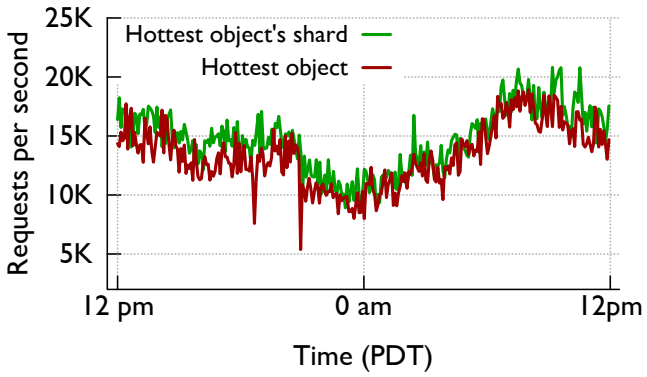


Figure 5.6: Traffic for hottest object.

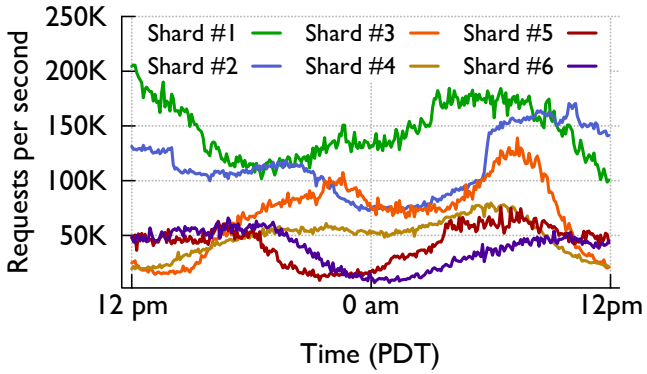


Figure 5.7: Traffic for hottest shards.

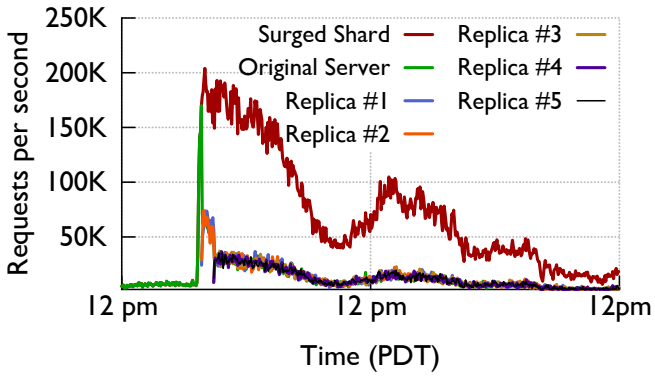


Figure 5.8: TAO reaction for surged shard.

much load on its replicas. Moreover, once all five replicas are created, they remain so situated for another 14 hours — even as the shard's popularity plummets to <50K rps, wasting significant cache memory. In Section 5.3, we further discuss the benefits and limitations of replication techniques in general and propose a potential improvement.

Problematic content placement. While consistent hashing is used in TAO to balance the mapping between data partitions (shards) and servers, studies of consistent hashing in other settings suggest that often the implementation of this mechanism is not sophisticated enough to overcome intrinsic issues with consistent hashing: insufficient rounds of hashing, low ratios between shards and servers, and poor choices of hashing functions. As a result, the shards may not be evenly distributed among different servers. To investigate the status of content placement in TAO and possible impact on load imbalance, we examined the correlation between the number of shards hosted by a server and the total traffic it serves.

Figure 5.9 shows that there is a strong relationship between the rank of cache servers by number of shards and their rank by traffic load (Spearman's $\rho = 0.848$, $p < 10^{-5}$). This correlation is especially true in the extreme cases: servers hosting the most shards tend to rank among the most-loaded ones, and the server hosting the fewest shards ranks within the least-loaded portion. This demonstrates that, for TAO, non-ideal shard placement plays an important role in causing load imbalance, in addition to content skewed popularity at the granularity of shards.

5.3 Mitigating Load Imbalance

We now use simulation to evaluate several possible techniques for mitigating load imbalance. In our comparison, we divide the approaches into two main categories: **(1)** consistent hashing, and **(2)** hot-content replication, which also includes front-end caching as a special case.

5.3.1 Trace Preparation

In order to properly evaluate traffic dynamics under different load-imbalance mitigating approaches, we use the ReqSample trace due to its high-resolution of real cache requests. The original trace is collected by the routing daemon on every Web front-end server, which randomly samples and records one per million Tao requests (so as to minimize measurement overhead on the live infrastructure). Cross-validation shows that the trace successfully captures all hot shards and hot objects contained in the TopShards aggregate trace. However, the sampling is too coarse to retain the traffic characteristics of every shard within a single cluster.

To cope with this problem, we treat this entire ReqSample trace as a trace for a single 'canonical silo' cluster that has been sampled at a higher frequency. This is feasible because **(1)** every Tao follower cluster is an independent caching deployment, and **(2)** graph queries to each Tao cluster behave similarly for popular content since Web requests are randomly distributed among all front-end clusters.

We verified these two properties on the TopShards by comparing the traffic dynamics of the top 1000 shards between a single cluster and the entire tier. The full-tier trace in ReqSample is effectively an aggregate of multiple clusters serving the same content (in different regions). Hence, we normalize the traffic to our canonical silo cluster based on that of one of the largest single clusters in the full trace. The manipulations on the entire trace yield the same normalized load distribution on the canonical silo cluster as originally found ($p < 10^{-5}$, Kolmogorov-Smirnoff test, $D = 0.2883$), except with higher sampling frequency.

5.3.2 Current Techniques

Our goal is twofold: **(1)** to understand how state-of-the-art approaches, for mitigating load skew on a distributed cache, complement one another on a real-world trace, and **(2)** to identify which mechanisms have opportunities for improvement. Two main classes of algorithms work in tandem to balance load: hashing schemes for balancing the *number* of shards allocated to servers, and replication schemes for balancing the *load* of the shards.

Hashing. The idea underpinning most partitioned services is that hashing identifiers to an abstract ring, and then dividing regions of the ring between servers, will roughly yield fair assignment of shards to servers. Assuming the ring state is maintained on every server, all lookups may be done locally — a property which facilitates distributed implementation — assuming the ring state can be kept up-to-date on all servers. As noted earlier, many existing systems leverage *consistent hashing* [55] which minimizes disruptions when servers are added or removed. Our experiments below include the popular open-source `libketama` library: a reference implementation of consistent hashing for memory caches [48].

Unfortunately, hashing schemes may impose significant skew on load distribution. Part of the reason is that these schemes do not consider traffic on shards. Yet even if all shards carry the same volume of traffic, the server with the highest load would still — with high probability — be responsible for twice as many shards as an average server [85].

Better distributed hashing schemes have yet to be found. One remedy for the lopsided number of shards is to divide the ring space further by hashing each server identi-

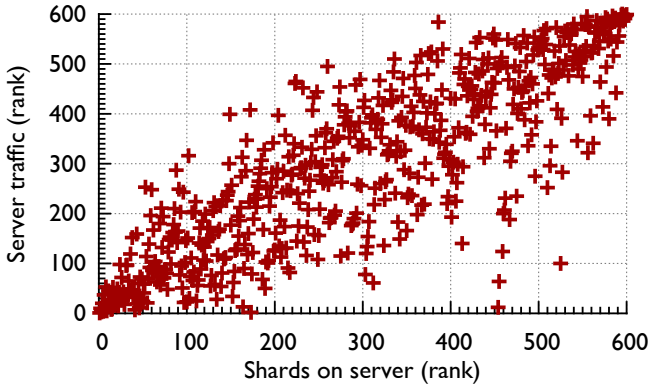


Figure 5.9: Load imbalance impact from shard placement.

fier many times onto the ring as "virtual nodes" [56, 30, 32]. Recently, Hwang and Wood proposed an adaptive hashing mechanism based on consistent hashing where the region boundaries of the ring space are dynamically adjusted according to load and cache hit rate on the servers to which they are assigned [46]. one must still address the problem of disparate traffic rates on shards.

To understand the opportunity for improving hashing mechanisms, we include a "*perfect hashing*" baseline in our simulations. In this baseline, a centralized controller ensures that all servers are responsible for exactly the same number of shards without concern for per-shard traffic.

Replication. We next add replication into the mix to combat the heavy-tailed load on shards. In a somewhat simplified summary, existing dynamic load-balancing techniques for distributed caching and storage systems operate in two phases: **(1)** *Detection*: identify hot servers and their hot contents, followed by **(2)** *Replication*: move data between servers to alleviate the high load, or divide the traffic across multiple servers by replicating hot contents elsewhere, sometimes on many nodes. We now survey several state-of-the-art detection and replication techniques.

A ***front-end cache*** is normally a small cache deployed in front of the tier experiencing the imbalanced load [34]. The detection phase depends on the replacement algorithm used by the cache, such as LRU (Least-Recently-Used), to detect very popular objects which exhibit high temporal locality. Once the hot object is detected, its replica is stored by the front-end cache and can serve all traffic flowing through the front-end via a local copy of the item without burdening other servers. Studies show that even small front-end caches can substantially alleviate skewed object access workload [**?**, 34].

Facebook's front-end Web servers already embed a small cache for popular TAO objects. Our traces, therefore, are focused on load imbalance *after* caches higher up in the hierarchy have been applied [**?**]. Moreover, our earlier analysis showed that after a layer of front-end caches, objects are no longer a significant factor in TAO's load-imbalance. Instead, the skew partly stems from the popularity of shards that each comprises multiple correlated objects connected through the social graph. However, compared to objects, shards are just too large to be cached at front-end Web servers: typical shard size in TAO is on the order of hundreds of megabytes, while typical object size is measured in kilobytes. Therefore, further improving the front-end cache is unlikely to resolve the load-imbalance situation in TAO — other solutions are needed.

Replicating hot content. Hong and Thettethodi [42] recently proposed augmenting the cache infrastructure to actively monitor and replicate hot objects across multiple servers. In their scheme, each `memcached` server monitors the popularity of its own content and informs clients about replication and rerouting decisions. The hotness-detection policy, in contrast to the "dominant" resource approach currently used in TAO, is implemented by maintaining a list of ranked counters, updated with an exponentially-weighted moving average for each item. Replication reconciliation is then controlled through time-based leases. Through simulations, we found that while this technique

reduces load imbalance caused by hot content, the reaction time can be shortened by improving the sensitivity of hot shard detection, and the memory overhead of replication can be reduced by adaptively checking the rank of the hot content in the load distribution.

As mentioned earlier, TAO's replication component monitors shard loads on servers and replicates dominant shards every ten minutes. Our analysis already showed that flash crowds and surges in popularity can destabilize an unlucky caching server within the 10-minute window. Moreover, its reverse routine is too conservative, resulting in unnecessary memory waste.

Streaming methods. There has been mounting interest in *streaming algorithms* which can process incoming data streams in a limited number of passes to provide approximate summaries of the data, including heavy-hitters and frequency estimation for popular items [28]. Frequency-estimation algorithms, useful for detecting hot shards, sample requests from the data stream, often at a very low rate, and carefully maintain a collection of candidates for hot shards. Frequency estimates can then be used to adaptively replicate shards based on their popularity. In an effort concurrent with ours, Hwang and Wood utilize streaming algorithms to reactively mitigate load balance [46].

Streaming algorithms have high performance, while requiring very small memory footprints and CPU overhead. They can be parallelized through sharding, much like a cache tier for scalability. The algorithms complement other solutions, such as front-end caching, and can be deployed transparently in an existing cache implementation.

Streaming algorithms can be faster than cache reports at detecting hot shards, as they operate at a finer temporal granularity and can identify trends practically in real-time. Cache reports are effectively snapshots over large time windows, whereas streaming algorithms can keep several summaries of shorter time intervals, thus providing a longer

and more detailed access history of popular shards. This can be further leveraged to identify patterns and make predictions on upcoming access frequencies. In our implementation, the most frequently requested shards in a 60-second time window are replicated to a constant number of servers. A shard is de-replicated when it has not been considered hot within the last four minutes. With a higher sampling ratio, hot shards can be identified at much finer granularity. Moreover, traffic estimates can be used to dynamically calculate the appropriate number of replicas for a given shard.

5.3.3 Comparison and Evaluation

We now evaluate the different hashing and replication schemes for load balancing on the trace from Facebook. We will define the $\frac{\max}{\text{avg}}$ statistic of a scheme to denote the volume of requests received on the most loaded server relative to an average server over the full trace (24 hours).

Figure 5.10 compares the impact of hashing schemes on load imbalance. Even with perfect hashing, the best load imbalance one obtains without using any replication method is a $\frac{\max}{\text{avg}}$ of 1.34, with the most loaded server 59% more loaded than the one with the lightest load. When no replication mechanism is used, Tao has $\frac{\max}{\text{avg}}$ of 1.46, slightly outperforming the `libketama` [48] reference consistent-hashing implementation. with $\frac{\max}{\text{avg}}$ of 1.52. If these methods also incorporated a perfect replication scheme, the difference is even more stark: with 41% more load on the highest loaded server than an average one for `libketama` compared to 17% for Tao. Finally, we also deduce that the hashing scheme within Tao can be improved by 8 percentage points, from $\frac{\max}{\text{avg}}$ of 1.25 to 1.17.

Figure 5.11 details how replication mechanisms affect load skew. We isolate the

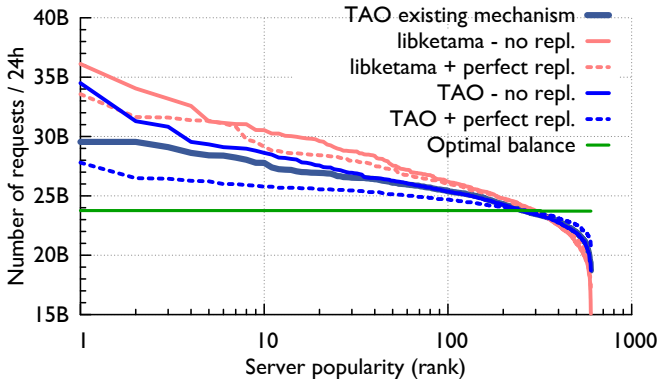


Figure 5.10: Cache load as hashing mechanism is modified.

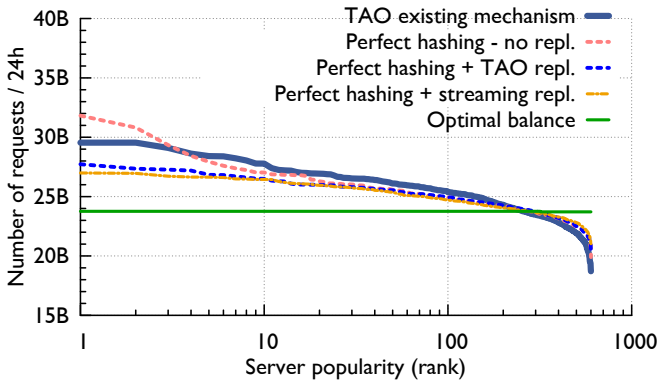


Figure 5.11: Cache load as replication mechanism is modified.

impact of the replication algorithm by assuming shards are uniformly distributed across servers. The streaming algorithm was significantly better at detecting hot shards than Tao's replication mechanism, but the overall reduction in load balance has room for improvement, moving $\frac{\max}{\text{avg}}$ from 1.17 to 1.14. Retaining Tao's current hashing mechanism, the graph shows an opportunity to improve on Tao's replication methods to decrease $\frac{\max}{\text{avg}}$ from 1.25 to 1.18. The streaming algorithm provided the most competitive replication scheme, but the detection and replication mechanisms can be substantially refined — part of ongoing work.

Takeaways. (1) Standard consistent hashing caused 52% more load on some servers relative to the average, and 239% relative to the least loaded server. Even if it were

coupled with an optimal replication algorithm, the most loaded server remains 41% more loaded than the average server. **(2)** TAO's hashing algorithm improved on plain consistent hashing, but the most loaded server still sustained 17% more load than the average server, and 34% more than the one with the lightest load, even if the replication scheme were to balance per-shard load perfectly. **(3)** Using streaming algorithms for hot-spot detection outperformed other replication schemes, but a $\frac{\max}{\text{avg}}$ of 1.14 suggests room for future improvement.

CHAPTER 6

FUTURE WORK & CONCLUSION

6.1 Open Questions for Future Work

While the research that forms this dissertation serves to advance our discipline's understanding and application of advanced caching solutions, additional unanswered questions remain that undoubtedly warrant continued investigation. In this section, we overview three major additional directions for building advanced caching services that apply quite universally to the modern Web-application domain.

Adaptive Caching Eviction. While we were in the process of analyzing Facebook's photo-serving stack and designing the flash-based caching framework, we observed that Facebook's workload continued to evolve. Therefore, even over this relatively short period of time (months to a year), the caching algorithm, with the best simulated performance, transitioned from the Segmented-LRU family to the Greedy-Dual-Size-Frequency family. Further investigation revealed the causes — (1) the ratio of various types of user-created contents keeps changing, and (2) the application logic that decides which content should be presented is not fixed. Such dynamics are not unique to Facebook, as any modern Web application that serves user-created content may experience the same phenomenon. This observation drove us to ask an important question: how to build a caching solution that always works efficiently given the workload it serves changes overtime.

An important area for future work that could solve this problem is designing adaptive caching algorithms, perhaps by monitoring the application logic and predicting the likelihood of future access based on meta information about the content currently in

service. Specifically, the caching policy in question can incorporate additional application signals beyond the frequency, recentness, and working set size of access; and can adaptively adjust the weight of different signals in deciding which content to cache. For instance, based on the likelihood that a type of content will be viewed by users within a certain geographic region and the application preference of serving low quality content within that region, the low quality content within that type could be cached with high priority at the specific location, or even be pre-fetched.

Pooling Strategy for Shared Caching. As caching has become ever more widely utilized, the types of data in caches have been diversified across an increasing number of Web applications. While some Web services only face this issue internally, the problem is more profound for multi-tenant, cloud-based Web hosting environments such as Amazon Web Service EC2 [1], where a plethora of independent Web sites may share the same caching infrastructure. Managing content generated by different applications in the same caching space is non-trivial: (1) rarely can a single caching algorithm work efficiently for all application workloads, and (2) performance isolation is hard to achieve. A common standard solution is pooling: the shared caching space is divided into independent cache pools, each of which is managed separately for a single type of content or a family of similar content. However, few studies have examined the optimal pooling strategy for shared caching, and no foundational principles have been enumerated to guide the pooling thresholds for given content types and associated workloads.

Load-aware Replica Placement. Ongoing work should focus on load-aware replica placement mechanisms, by leveraging the load-balance analysis contained in this dissertation. Particularly for in-memory distributed caching systems, the number of replicas and their placement in cache highly affects the load-balance status across the caching

tier. Furthermore, to maintain bounded data-fetching latency as well as tunable network overhead in a multi-layered caching deployment, we must address the questions of within which layer the content should reside, and at what time its placement should be changed.

6.2 Conclusion

Both in-memory caching and static-content caching serve as critical components for the scalability and efficiency of modern Web applications. The design of an advanced caching infrastructure for the modern Web exposes challenges from three dimensions: non-traditional workloads, hardware limitations of adopting advanced caching algorithms, and load imbalances caused by disparate content popularities. This dissertation provides three studies that are based on Facebook's caching systems; each aims to address a different challenge.

To analyze the modern Web-application workload and its impact on content caching design, we instrumented the entire Facebook photo-serving stack, thereby obtaining traces representative of Facebook's full workload. To the best of our knowledge, this is the first study to examine an entire Internet image-serving infrastructure at a massive scale. And a number of valuable findings emerge from this integrated perspective — workload pattern, traffic distribution, and geographic system dynamics — yielding insights helpful to future system designers. Specifically, we quantified the content popularity shift layer by layer and its impact on different caching tiers; this result (1) provides a new perspective of content-serving workload that researchers can benefit from, and (2) is valuable for performance estimation of a different capacity and algorithm setup within similar application workload. We found that nearly half the content-fetch-

ing traffic traveled remotely for network engineering reasons, and as a consequence the perceived latency benefit of operating Edge caches independently was detrimented. This counter-intuitive finding points to a possible caching option that benefits distributed caching infrastructures running on top of today's Internet: it may be worthwhile to explore collaborative caching at geographic (nationwide or global) scales. Our analysis also examined the relationship between content access and associated meta-data in the application domain. It was part of a larger effort in finding more advanced caching algorithms that outperform our currently popular but simple solutions, such as LRU, for modern Web applications. Our results proves that Segmented-LRU has a large potential hit ratio gain at both Edge and Origin layers, and there is still a large improvement space for future caching algorithm designers.

Flash memory, with its high capacity, high IOPS, and complex performance characteristics, poses new opportunities and challenges for implementing advanced static-content caching. In this dissertation we have also presented two frameworks, RIPQ and SIPQ, that implement approximate priority queues efficiently on flash and thus support the implementation of advanced caching algorithms. To the best of our knowledge, both RIPQ and SIPQ are novel designs in this underexplored area. As the adoption of modern flash devices becomes increasingly common in the modern Web caching stack, the benefits from RIPQ and SIPQ would be more appreciated. Through evaluation with Facebook photo traces, we demonstrated that RIPQ achieves high hit ratio fidelity, high throughput, and low write amplification for all algorithms tested, with a moderate memory consumption of 5GB. Although the memory overhead of RIPQ is small for a typical server, it still prevents a broader adoption for memory constrained environments. Our simplified RIPQ with a single insertion point, SIPQ, provides a memory-saving alternative with good results for simple algorithms like LRU. One important area of study that deserves further investigation is reducing RIPQ's memory consumption. For example,

with newer generation of flash devices that expose their internal device structures, RIPQ has an opportunity to arrange its restricted insertion points in a device friendly manner so that the buffering size for maximal write throughput can be greatly reduced.

The scalability of today's popular web sites is also enabled by large clusters of in-memory cache servers. Each server in a cluster must be equipped to handle peak load, but this implies extensive overprovisioning due to load imbalance across the cache servers. We investigated the causes of the load skew on real-world traces from Facebook's TAO cluster, and demonstrated that imbalanced content placement and popularity disparity caused by dependent sharding both play major roles, while the impact from object level popularity skewness is not significant. As we mentioned earlier, the increasing complexity of data types and operation types that a cache solution needs to support has imposed additional constraints on its data partitioning manner. As a result, load balancing can no longer to achieved by random spreading out the data and needs alternative solutions. Through simulation, we find current load-balancing techniques — including consistent hashing, as well as different flavors of replication — only partially address such skew, while a streaming-analytics based approach holds promise for further improvement. The major benefit of streaming-analytics comes from its low cost: it is able to detect temporally popular content with less than 1% of the cache request traffic, and therefore can be operated continuously without interruption. Future works on streaming analytics based approach could take the advantage of observing high fidelity requests and apply more complicated policies in predicting popular content proactively. Our current results pave the way for continued research into more effective mitigation techniques for load skew, which would reduce infrastructure requirements and improve cache performance.

BIBLIOGRAPHY

[1] Amazon Web Service. http://aws.amazon.com/ec2/, 2014.

[2] Apache HTTP Server Project. http://httpd.apache.org/, 2014.

[3] HHVM. http://hhvm.com/, 2014.

[4] LevelDB, A fast and lightweight key/value database library by Google. https://code.google.com/p/leveldb, 2014.

[5] Memcached. http://memcached.org/, 2014.

[6] Nginx. http://nginx.com/, 2014.

[7] RocksDB, A persistent key-value store for fast storage environments. http://rocksdb.org, 2014.

[8] Marc Abrams, Charles R. Standridge, Ghaleb Abdulla, Edward A. Fox, and Stephen Williams. Removal Policies in Network Caches for World-Wide Web Documents. In *Proc. Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, pages 293–305, Palo Alto, California, 1996. ACM.

[9] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. Design Tradeoffs for SSD Performance. In *Proc. Annual Technical Conference (ATC)*, pages 57–70, Boston, MA, 2008. USENIX Association.

[10] Alfred V. Aho, John E. Hopcroft, and Jeffrey Ullman. *Data Structures and Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1st edition, 1983.

[11] Ashok Anand, Chitra Muthukrishnan, Steven Kappes, Aditya Akella, and Suman Nath. Cheap and Large CAMs for High Performance Data-intensive Networked Systems. In *Proc. Conference on Networked Systems Design and Implementation (NSDI)*, pages 29–29, San Jose, California, 2010. USENIX Association.

[12] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: A Fast Array of Wimpy Nodes. In *Proc. Symposium on Operating Systems Principles (SOSP)*, pages 1–14, Big Sky, Montana, 2009. ACM.

[13] David G. Andersen and Steven Swanson. Rethinking Flash in the Data Center. *IEEE Micro*, 30(4):52–54, July 2010.

[14] Anirudh Badam, KyoungSoo Park, Vivek S. Pai, and Larry L. Peterson. HashCache: Cache Storage for the Next Billion. In *Proc. Symposium on Networked Systems Design and Implementation (NSDI)*, pages 123–136, Boston, MA, 2009. USENIX Association.

[15] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a Distributed File System. In *Proc. Symposium on Operating Systems Principles (SOSP)*, Pacific Grove, California, 1991. ACM.

[16] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. Finding a Needle in Haystack: Facebook's Photo Storage. In *Proc. Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, BC, Canada, October 2010. USENIX Association.

[17] Armond Bigian. Blobstore: Twitter's in-house photo storage system. `http://tinyurl.com/cda5ahq`, 2012.

[18] Luc Bouganim, Björn Jónsson, and Philippe Bonnet. uflip: Understanding flash io patterns. *arXiv preprint arXiv:0909.1780*, 2009.

[19] Lee Breslau, Pei Cue, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web Caching and Zipf-like Distributions: Evidence and Implications. In *Proc. International Conference on Computer Communications (INFOCOM)*. IEEE, 1999.

[20] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. TAO: Facebook's Distributed Data Store for the Social Graph. In *Proc of Conference on Annual Technical Conference (ATC)*, pages 49–60, San Jose, CA, 2013. USENIX Association.

[21] Michael Butkiewicz, Harsha V. Madhyastha, and Vyas Sekar. Understanding website complexity: measurements, metrics, and implications. In *Proc. SIGCOMM Internet Measurement Conference (IMC)*, Berlin, Germany, November 2011. ACM.

[22] Pei Cao and Sandy Irani. Cost-aware WWW Proxy Caching Algorithms. In *Proc. Symposium on Internet Technologies and Systems (USITS)*, pages 18–18, Monterey, California, 1997. USENIX Association.

[23] William Chan. Chromium cache metrics. `http://tinyurl.com/csu34wa`, 2013.

[24] Bernard Chazelle. The Soft Heap: An Approximate Priority Queue with Optimal Error Rate. *Journal of the ACM (JACM)*, 47(6):1012–1027, November 2000.

[25] Yanpei Chen, Kiran Srinivasan, Garth Goodson, and Randy Katz. Design Implications for Enterprise Storage Systems via Multi-Dimensional Trace Analysis. In *Proc. Symposium on Operating Systems Principles (SOSP)*, Cascais, Portugal, 2011. ACM.

[26] Ludmila Cherkasova and Gianfranco Ciardo. Role of Aging, Frequency, and Size in Web Cache Replacement Policies. In *Proc. High-Performance Computing and Networking (HPCN)*, pages 114–123, Amsterdam, The Netherlands, 2001. Springer.

[27] Hong-Tai Chou and David J. DeWitt. An Evaluation of Buffer Management Strategies for Relational Database Systems. In *Proc. International Conference on Very Large Data Bases (VLDB)*, pages 127–141, Stockholm, Sweden, 1985. VLDB Endowment.

[28] Graham Cormode and Marios Hadjieleftheriou. Methods for Finding Frequent Items in Data Streams. *The VLDB Journal*, 19(1):3–20, February 2010.

[29] LSI Corporation. *LSI Nytro XP6210 Spec*, 2014 (accessed April 25, 2014).

[30] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area Cooperative Storage with CFS. In *Proc. Symposium on Operating Systems Principles (SOSP)*, pages 202–215, Banff, Alberta, Canada, 2001. ACM.

[31] Jeffrey Dean. Challenges in Building Large-scale Information Retrieval Systems: Invited Talk. In *Proc. International Conference on Web Search and Data Mining (WSDN)*, pages 1–1, Barcelona, Spain, 2009. ACM.

[32] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *Proc. Symposium on Operating Systems Principles (SOSP)*, pages 205–220, Stevenson, Washington, 2007. ACM.

[33] Robert Escriva, Bernard Wong, and Emin Gün Sirer. HyperDex: A Distributed, Searchable Key-value Store. In *Proc. Conference on Applications, Technologies,*

Architectures, and Protocols for Computer Communication (SIGCOMM), pages 25–36, Helsinki, Finland, 2012. ACM.

[34] Bin Fan, Hyeontaek Lim, David G. Andersen, and Michael Kaminsky. Small Cache, Big Effect: Provable Load Balancing for Randomly Partitioned Cluster Services. In *Proc. Symposium on Cloud Computing (SOCC)*, pages 23:1–23:12, Cascais, Portugal, 2011. ACM.

[35] Michael L Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34(3):596–615, 1987.

[36] Michael J. Freedman. Experiences with CoralCDN: A Five-Year Operational View. In *Proc. Symposium on Networked Systems Design and Implementation (NSDI 10)*, San Jose, CA, April 2010. USENIX Association.

[37] Michael J. Freedman, Eric Freudenthal, and David Mazières. Democratizing content publication with Coral. In *Proc. Symposium on Networked Systems Design and Implementation (NSDI)*, San Francisco, California, March 2004. USENIX Association.

[38] Alex Gartrell, Mohan Srinivasan, Bryan Alger, and Kumar Sundararajan. McDipper: A key-value cache for Flash storage. `http://tinyurl.com/c39w465`, 2013.

[39] Krishna P. Gummadi, Richard J. Dunn, Stefan Saroiu, Steven D. Gribble, Henry M. Levy, and John Zahorjan. Measurement, Modeling, and Analysis of a Peer-to-Peer File-Sharing Workload. In *Proc. Symposium on Operating Systems Principles (SOSP)*, Bolton Landing, NY, December 2003. ACM.

[40] Lei Guo, Enhua Tan, Songqing Chen, Zhen Xiao, and Xiaodong Zhang. The stretched exponential distribution of internet media access patterns. In *Proc. Symposium on Principles of distributed computingi (PODC)*, Toronto, Canada, August 2008. ACM.

[41] Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. A File is Not a File: Understanding the I/O Behavior of Apple Desktop Applications. In *Proc. Symposium on Operating Systems Principles (SOSP)*, Cascais, Portugal, 2011. ACM.

[42] Yu-Ju Hong and Mithuna Thottethodi. Understanding and Mitigating the Impact of Load Imbalance in the Memory Caching Tier. In *Proc. Symposium on Cloud Computing (SOCC)*, pages 13:1–13:17, Santa Clara, California, 2013. ACM.

[43] Xiao-Yu Hu, Evangelos Eleftheriou, Robert Haas, Ilias Iliadis, and Roman Pletka. Write Amplification Analysis in Flash-based Solid State Drives. In *Proc. The Israeli Experimental Systems Conference (SYSTOR)*, pages 10:1–10:9, Haifa, Israel, 2009. ACM.

[44] Qi Huang, Ken Birman, Robbert van Renesse, Wyatt Lloyd, Sanjeev Kumar, and Harry C. Li. An Analysis of Facebook Photo Caching. In *Proc. Symposium on Operating Systems Principles (SOSP)*, pages 167–181, Farminton, Pennsylvania, 2013. ACM.

[45] Sai Huang, Qingsong Wei, Jianxi Chen, Cheng Chen, and Dan Feng. Improving flash-based disk cache with Lazy Adaptive Replacement. In *Proc. Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, Long Beach, CA, 2013. IEEE.

[46] Jinho Hwang and Timothy Wood. Adaptive Performance-Aware Distributed Memory Caching. In *Proc. International Conference on Autonomic Computing (ICAC)*, pages 33–43, San Jose, CA, 2013. USENIX Association.

[47] Sunghwan Ihm and Vivek S. Pai. Towards Understanding Modern Web Traffic. In *Proc. SIGCOMM Internet Measurement Conference (IMC)*, Berlin, Germany, November 2011. ACM.

[48] Richard James. `libketama`: a consistent hashing algo for memcache clients. `http://github.com/RJ/ketama`, Apr 2007.

[49] Song Jiang and Xiaodong Zhang. LIRS: An Efficient Low Inter-reference Recency Set Replacement Policy to Improve Buffer Cache Performance. In *Proc. International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 31–42, Marina Del Rey, California, 2002. ACM.

[50] Shudong Jin and Azer Bestavros. Temporal Locality in Web Request Streams: Sources, Characteristics, and Caching Implications. Technical report, Boston, MA, 1999.

[51] Robert Johnson. Facebook's Scribe technology now open source. `http://tinyurl.com/d7qzest`, 2008.

[52] Jaeyeon Jung, Balachander Krishnamurthy, and Michael Rabinovich. Flash crowds and Denial of Service attacks: Characterization and implications for CDNs and web sites. In *Proc. International World Wide Web conference (WWW)*, Honolulu, Hawaii, May 2002. ACM.

[53] Krishna Kant. Data center evolution: A tutorial on state of the art, issues, and challenges. *Elsevier Computer Networks*, 53(17):2939–2965, 2009.

[54] Ramakrishna Karedla, J Spencer Love, and Bradley G Wherry. Caching Strategies to Improve Disk System Performance. *Computer*, 27(3):38–46, 1994.

[55] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proc. Annual Symposium on Theory of Computing (STOC)*, pages 654–663, El Paso, Texas, 1997. ACM.

[56] David Karger, Alex Sherman, Andy Berkheimer, Bill Bogstad, Rizwan Dhanidina, Ken Iwamoto, Brian Kim, Luke Matkins, and Yoav Yerushalmi. Web Caching with Consistent Hashing. In *Proc. International Conference on World Wide Web (WWW)*, pages 1203–1213, Toronto, Canada, 1999. Elsevier North-Holland, Inc.

[57] Bryan Kate, Eddie Kohler, Michael S. Kester, Neha Narula, Yandong Mao, and Robert Morris. Easy Freshness with Pequod Cache Joins. In *Proc. Conference on Networked Systems Design and Implementation (NSDI)*, pages 415–428, Seattle, WA, 2014. USENIX Association.

[58] Hyojun Kim and Seongjun Ahn. BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage. In *Conference on File and Storage Technologies (FAST)*, San Jose, California, 2008. USENIX.

[59] Hyojun Kim, Moonkyung Ryu, and Umakishore Ramachandran. What is a Good Buffer Cache Replacement Scheme for Mobile Flash Storage? In *Proc. of the SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, London, England, UK, 2012. ACM.

[60] Michael Kirkland. Flashcache at Facebook: From 2010 to 2013 and beyond. `http://tinyurl.com/oljloxb`, 2013.

[61] Sang-Won Lee, Dong-Joo Park, Tae-Sun Chung, Dong-Ho Lee, Sangwon Park, and Ha-Joo Song. A log buffer-based flash translation layer using fully-associative sector translation. *ACM Transactions on Embedded Computing Systems (TECS)*, 6(3):18, 2007.

[62] Sungjin Lee, Dongkun Shin, Young-Jin Kim, and Jihong Kim. Last: Locality-aware sector translation for nand flash memory-based storage systems. *ACM SIGOPS Operating Systems Review*, 42(6):36–42, October 2008.

[63] Xiaozhou Li, David G. Anderson, Michael Kaminsky, and Michael J Freedman. Algorithmic Improvements for Fast Concurrent Cuckoo Hashing. In *Proc. European Conference on Computer Systems (EuroSys)*, pages 27:1–27:14, Amsterdam, The Netherlands, 2014. ACM.

[64] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. SILT: A memory-efficient, high-performance key-value store. In *Proc. Symposium on Operating Systems Principles (SOSP)*, pages 1–13, Cascais, Portugal, 2011. ACM.

[65] Silvano Maffeis. Cache Management Algorithms for Flexible Filesystems. *ACM SIGMETRICS Performance Evaluation Review*, 21(2):16–25, 1993.

[66] Nimrod Megiddo and Dharmendra S. Modha. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *Proc. Conference on File and Storage Technologies (FAST)*, pages 115–130, San Francisco, CA, 2003. USENIX Association.

[67] Changwoo Min, Kangnyeon Kim, Hyunjin Cho, Sang-Won Lee, and Young Ik Eom. SFS: Random Write Considered Harmful in Solid State Drives. In *Proc. Conference on File and Storage Technologies (FAST)*, San Jose, CA, 2012. USENIX Association.

[68] Michael Mitzenmacher. The Power of Two Choices in Randomized Load Balancing. *IEEE Transactions on Parallel Distributed Systems*, 12(10):1094–1104, October 2001.

[69] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling Memcache at Facebook. In *Proc. Conference on Networked Systems Design and Implementation (NSDI)*, pages 385–398, Lombard, IL, 2013. USENIX Association.

[70] John K. Ousterhout, Hervé Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson. A Trace-Driven Analysis of the UNIX 4.2 BSD File System. In *Proc. Symposium on Operating Systems Principles (SOSP)*, Orcas Island, Washington, 1985. ACM.

[71] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. SDF: software-defined flash for web-scale internet storage systems. In *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 471–484. ACM, 2014.

[72] Patrick ONeil, Edward Cheng, Dieter Gawlick, and Elizabeth ONeil. The log-

structured merge-tree (LSM-tree). *Springer Acta Informatica*, 33(4):351–385, 1996.

[73] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Elsevier Journal of Algorithms*, 51(2):122–144, 2004.

[74] Seon-yeong Park, Dawoon Jung, Jeong-uk Kang, Jin-soo Kim, and Joonwon Lee. CFLRU: A Replacement Algorithm for Flash Memory. In *Proc. International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, Seoul, Korea, 2006. ACM.

[75] David A. Patterson, Garth Gibson, and Randy H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proc. International Conference on Management of Data (SIGMOD)*, Chicago, Illinois, 1988. ACM.

[76] Mendel Rosenblum and John K Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, 1992.

[77] Stefan Saroiu, Krishna P. Gummadi, Richard J. Dunn, Steven D. Gribble, and Henry M. Levy. An analysis of Internet Content Delivery Systems. In *Proc. Symposium on Operating System Design and Implementation (OSDI)*, Boston, MA, December 2002. USENIX.

[78] Salvatore Scellato, Cecilia Mascolo, Mirco Musolesi, and Jon Crowcroft. Track Globally, Deliver Locally: Improving Content Delivery Networks by Tracking Geographic Social Cascades. In *Proc. International World Wide Web conference (WWW)*, Hyderabad, India, March 2011. ACM.

[79] Peter Scheuermann, Gerhard Weikum, and Peter Zabback. Data Partitioning and Load Balancing in Parallel Disk Systems. *The VLDB Journal*, 7(1):48–66, February 1998.

[80] Radu Stoica, Manos Athanassoulis, Ryan Johnson, and Anastasia Ailamaki. Evaluating and Repairing Write Performance on Flash Devices. In *Proc. International Workshop on Data Management on New Hardware (DaMoN)*, pages 9–14, Providence, Rhode Island, 2009. ACM.

[81] Swaroop Kavalanekar and Bruce L. Worthington and Qi Zhang and Vishal Sharda. Characterization of storage workload traces from production Windows Servers. In *Proc. International Symposium on Workload Characterization (IISWC)*, Seattle, Washington, 2008. IEEE.

[82] Ashish Thusoo. Hive - A Petabyte Scale Data Warehouse using Hadoop. `http://tinyurl.com/bprpy5p`, 2009.

[83] Cristian Ungureanu, Biplob Debnath, Stephen Rago, and Akshat Aranya. TBF: A memory-efficient replacement policy for flash-based caches. In *Proc. International Conference on Data Engineering (ICDE)*, pages 1117–1128, Brisbane, Australia, 2013. IEEE.

[84] Werner Vogels. File system usage in Windows NT 4.0. In *Proc. Symposium on Operating Systems Principles (SOSP)*, Charleston, South Carolina, 1999. ACM.

[85] Xiaoming Wang and Dmitri Loguinov. Load-balancing Performance of Consistent Hashing: Asymptotic Analysis of Random Node Join. *ACM/IEEE Transactions on Networking (TON)*, 15(4):892–905, August 2007.

[86] Patrick Wendell and Michael J. Freedman. Going viral: flash crowds in an open CDN. In *Proc. SIGCOMM Internet Measurement Conference (IMC)*, Berlin, Germany, November 2011. ACM.

[87] Roland P. Wooster and Marc Abrams. Proxy Caching That Estimates Page Load Delays. In *Proc. International Conference on World Wide Web*, pages 977–986, Santa Clara, California, 1997. Elsevier Science Publishers Ltd.

[88] Neal Young. The K-server Dual and Loose Competitiveness for Paging. *Springer Algorithmica*, 11(6):525–541, 1994.

[89] Yuanyuan Zhou, James Philbin, and Kai Li. The Multi-Queue Replacement Algorithm for Second Level Buffer Caches. In *Proc. Annual Technical Conference (ATC)*, pages 91–104, Boston, MA, 2001. USENIX Association.

[90] Timothy Zhu, Anshul Gandhi, Mor Harchol-Balter, and Michael A. Kozuch. Saving Cash by Using Less Cache. In *Proc. Conference on Hot Topics in Cloud Ccomputing (HotCloud)*, pages 3–3, Boston, MA, 2012. USENIX Association.