Data-driven Performance Modeling in Complex Networked Systems

By

Yazhuo Zhang
Doctor of Philosophy

Computer Science

Ymir Vigfusson, Ph.D.
Advisor

Avani Wildani, Ph.D.
Committee Member

Nosayba El-Sayed, Ph.D.
Committee Member

Rebecca Isaacs, Ph.D.
Committee Member

Accepted:

Kimberly Jacob Arriola, Ph.D.
Dean of the James T. Laney School of Graduate Studies

Date

Data-driven Performance Modeling in Complex Networked Systems

By

Yazhuo Zhang
B.S., South China University of Technology, Guangzhou, 2016
M.S., South China University of Technology, Guangzhou, 2019

Advisor: Ymir Vigfusson, Ph.D.

An abstract of
A dissertation submitted to the Faculty of the
James T. Laney School of Graduate Studies of Emory University
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy
in Computer Science
2024

Abstract

Data-driven Performance Modeling in Complex Networked Systems
By Yazhuo Zhang

Complex networked systems are a ubiquitous presence in our daily lives, but these systems require active maintenance and management. Their reliability and efficiency hinge on operators being able to reason about capacity planning, bottleneck identification, and making informed decisions about scaling, load balancing, and system optimization. However, it is challenging to perform performance modeling for complex networked systems because of a multitude of challenges, including the complexity of the components themselves, the intricate dependencies that exist between them, and a gigantic configuration space that must be navigated.

This dissertation focuses on how to conduct data-driven performance modeling in complex networked systems, guided by two main principles. The first principle is to decompose the entire system into key components that have interpretable interactions. The second principle is to leverage empirical system data to inform the modeling process and to guide the decision making.

Using these two key principles, we conduct data-driven performance modeling in three distinct types of general complex systems: microservice-based applications, large-scale web cache systems, and content delivery networks (CDNs). We present LatenSeer, a data-driven modeling framework for estimating end-to-end latency distributions in microservice-based web applications. By leveraging distributed tracing data, LatenSeer models the latency experienced by end users at scale, in an effective, accurate, and robust manner. Next, we discuss SIEVE, an cache eviction primitive, inspired by real-world web cache workloads and informed by data on cache item access patterns. SIEVE is simpler than LRU and provides better efficiency and scalability than state-of-the-art algorithms. Finally, we introduce THEODON, a framework for modeling CDN architectures via modular simulations, enabling fast discovery of efficient architectures and parameters configurations that balance performance and cost.

Data-driven Performance Modeling in Complex Networked Systems

By

Yazhuo Zhang
B.S., South China University of Technology, Guangzhou, 2016
M.S., South China University of Technology, Guangzhou, 2019

Advisor: Ymir Vigfusson, Ph.D.

A dissertation submitted to the Faculty of the
James T. Laney School of Graduate Studies of Emory University
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy
in Computer Science
2024

Acknowledgments

Pursuing a PhD has been an incredible journey and I am deeply grateful for the support and guidance from my mentors, as well as the enduring encouragement from my family and friends throughout this journey.

First and foremost, I would like to extend my deepest gratitude to my advisor, Ymir Vigfusson, for not just guiding me along the research path, but for being supportive all the time. Ymir's exceptional skills in system building and mathematical thinking, combined with a broad and visionary perspective, have been instrumental in shaping my approach to research. I am truly fortunate to have had Ymir as a mentor, whose influence has been pivotal in my development as a researcher.

I would like to thank Rebecca Isaacs and Yao Yue for their excellent advice, which helped me find my way at critical crossroads in the PhD journey. Rebecca teaches me how to systematically solve problems, and she always give honest, wise advice. Yao teaches me how to think broadly and ask the right questions. Both have provided me with invaluable insights, and their guidance has been a cornerstone of my growth. Thank you to Avani Wildani and Nosayba El-Sayed for or their support and for being on my dissertation committee.

I am fortunate to have brilliant collaborators in academia and industry on the work described in this dissertation. I'm especially thankful to work with Juncheng Yang for his patient guidance. Juncheng teaches me how to write a good paper, how to present work, and more importantly being patient to fresh researchers. Thank you to Rashmi Vinayak, Lei Zhang, Irfan Ahmad, Omer Majeed, Khubaib Umer, Andy Banta, Ziyue Qiu, and Jonathan Mace for their valuable contributions to this work.

My time with the SimBioSys research group has been nothing short of remarkable. I am grateful for the opportunity to work and spend time with each member: Gary Vestal, Vishwanath Seshagiri, Mark Ma, Shrey Gupta, Si Chen, and Pranav Bhandari. The board game sessions with you all will be dearly missed and always cherished.

The journey through my PhD has been made all the more fulfilling thanks to the presence of wonderful friends. I am deeply thankful to Yanan and Ziye for being incredible sources of positive energy and fun. Thank to Xi for consistent support, and I will remember our adventures exploring this country together. A special thank you to Han, my amazing friend and roommate for four years, for the mutual support we've shared. I am incredibly grateful to have Hongyang as such a close friend, even when we are thousands of miles apart. Thank you to Wenjing, Hejie, Jiaying, Xuan, Ziwei, and many others for the great memories together at Emory.

I am very grateful to Qingmiao for having been my pillar of support, enabling me to bravely face challenges. Your companionship was invaluable to me at a time when I was naive, immature, and timid, helping my growth into the truly independent individual I have become.

Lastly, my deepest appreciation goes to my family, especially my parents. Their endless encouragement and support have been the bedrock of my journey.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Complex networked systems pervade our daily lives, ranging from social media platforms like Twitter to e-commerce giants such as Amazon, and cloud service providers like Microsoft Azure, where distributed systems form the core of large-scale applications. Performance modeling for these systems is a vital task for system operators, encompassing the simulation and analysis of how these systems behave under various conditions. This involves understanding and predicting factors like latency, throughput, and resource utilization across different components. Crucial for capacity planning, identifying potential bottlenecks, and making informed decisions about scaling, load balancing, and system optimization, performance modeling is key to maintaining system reliability and efficiency [53, 105]. However, performance modeling is becoming more challenging as these systems grow more complex and face increasingly variable traffic patterns.

Performance modeling in complex networked systems is difficult for several reasons. First, these systems typically are composed of numerous interconnected components with intricate dependencies, leading to a cascading effect on performance. For instance, microservice-based applications such as Twitter or Uber consist of hundreds of interdependent microservices, making performance estimation a significant challenge [57, 90].

Additionally, the components within these large-scale systems are becoming increasingly complicated. Take, for example, cache systems, which are vital components in these environments. They are evolving with the integration of advanced, ML-based cache eviction algorithms, leading to increased unpredictability, system overhead, and not always optimal performance [33, 170, 202]. Moreover, large-scale systems often have an inherently vast configuration space. Content Delivery Networks (CDNs), for example, involve configurations that include the number of machines, the type of hardware used, the capacity of each machine, and more [2, 13, 169]. Even minor changes in these parameters can dramatically impact overall system performance [214].

Traditional performance modeling methods, such as queuing theory models and analytical model [117], are foundational yet often inadequate for complex networked systems. They tend to be static, relying on fixed parameters and assumptions. However, networked systems like microservices or CDNs are dynamic, with continually changing workloads and configurations. Traditional models typically fail to capture the nuanced complexities of modern networked systems and usually concentrate on individual system components, resulting in a limited scope for making informed, system-wide performance improvement decisions.

Fortunately, there are promising opportunities for more effectively modeling the performance of these complicated networked environments. The availability of detailed data paves the way for more dynamic and accurate modeling approaches. For example, distributed tracing systems in microservice or serverless architectures offer extensive insights into service interactions and latencies, allowing for the development of more precise models that reflect the intricate and evolving nature of these systems [102, 151, 178, 244]. Similarly, in large-scale storage systems, analyzing workloads and access patterns can lead to more effective resource allocation and caching strategies [222, 227]. However, we must identify the most suitable system abstraction that capitalizes on the available data for modeling performance. Hence, the overarching goal is to find a balance that provides a comprehensive system overview while enabling informed decision-making.

To better model the complex networked systems, we have two main requirements. First, we need an appropriate abstraction of the entire system under study. Rather than delving into every minute detail, we would like to focus on key components that accurately epitomize the system's overall functioning. This level of abstraction needs to encapsulate the fundamental aspects of the system's functionality and interactions, thereby simplifying the analysis and yielding more insightful results. Second, the abstraction should facilitate effective performance analysis. The use of empirical data is necessary to inform the modeling process and to guide the decision making.

1.1 Contributions

The key question this dissertation explores is *how should we model performance in complex networked systems?* We address this question using two main principles. The first principle is to decompose the entire system into key components with interpretable interactions. The major challenge is enabling enabling a holistic understanding of the system while managing the complexity of individual components and their interdependencies. The second principle is to leverage empirical system data to inform modeling process and to guide the decision making. We explore to use data-driven approaches for making informed decisions to enhance system efficiency, whether in terms of latency or resource utilization.

Using these two key principles, we conduct data-driven performance modeling across three representative systems: microservice-based application, large-scale web caching systems, and CDNs. Below, we provide a brief overview of each research project.

LatenSeer: Causal Modeling of End-to-End Latency Distributions by Harnessing Distributed Tracing

End-to-end latency estimation in web applications is crucial for system operators to foresee the effects of potential changes, helping ensure system stability, optimize cost, and improve user experience. However, estimating latency in microservices-based architectures is challenging due to the complicated interactions between hundreds or thousands of loosely coupled microservices [231]. Current approaches either track only latency-critical paths or require laborious bespoke instrumentation, which is unrealistic for end-to-end latency estimation in microservices-based systems.

We present LatenSeer, a data-driven modeling framework for estimating end-to-end latency distributions in microservice-based web applications. LatenSeer proposes novel data structures to accurately represent causal relationships between services, overcoming the drawbacks of simple dependency representations that fail to capture the complexity of microservices. LatenSeer leverages distributed tracing data to practically and accurately model end-to-end latency at scale. It enables developers to explore what-if scenarios to debug and improve the performance of their applications. Our evaluation shows that LatenSeer predicts latency within a 5.35% error, outperforming the state-of-the-art that has an error rate of more than 9.5%.

LatenSeer is highly adoptable as it requires no changes to the existing applications with pure usage of distributed tracing data. The development and validation of LatenSeer are grounded in real-world application, utilizing production services and tracing data from Twitter. Additionally, the system is accessible as an open-source platform.

Sieve: an Efficient Turn-Key Eviction Algorithm for Web Caches

Web caching systems are essential components of modern Internet infrastructure, including microservice-based applications. Cache is a component that uses fast but expensive medium to store data, so future request for that data can be served faster. Because the high cost of this storage medium, cache usually has very limited space. Therefore, deciding what data to be stored in the cache is very important. The core for the cache performance is eviction algorithm, which decides which object to evict when the cache is full. The primary focus of an eviction algorithm is to maximize efficiency by minimizing cache misses. However, despite the development of numerous eviction algorithms in recent decades, these tend to become increasingly complicated, often yielding only minor improvements [232]. We apply the second principle about data-driven modeling. This involves analyzing real-world web cache workloads and leveraging insights from cache item access patterns. Such an approach simplifies the eviction process and boosts efficiency, moving beyond traditional methods to optimize this crucial component within caching infrastructure.

We present SIEVE, a cache primitive that is simpler than LRU and provides better than state-of-the-art efficiency and scalability for web cache workloads. SIEVE's development and refinement are informed by real production web cache workloads, leveraging data on access patterns – specifically, the frequency of cache item visits – to guide strategic decision-making. We implemented SIEVE in five production cache libraries, requiring fewer than 20 lines of code changes on average. Our evaluation on 1559 cache traces from 7 sources shows that SIEVE achieves up to 63.2% lower miss ratio than ARC. Moreover, SIEVE has a lower miss ratio than 9 state-of-the-art algorithms on more than 45% of the 1559 traces, while the next best algorithm only has a lower miss ratio on 15%. SIEVE's simplicity comes with superior scalability as cache hits require no locking. Our prototype achieves twice the throughput of an optimized 16-thread LRU implementation. SIEVE is more than an eviction algorithm; it can be used as a cache primitive to build advanced eviction algorithms just like FIFO and LRU.

SIEVE is both simple and efficient to be adopted in production, and is already having impact. Several popular caching libraries have integrated SIEVE, such as golang-fifo [12] and Ristretto [14]. SIEVE is also adopted in multiple production system, such as DragonFly [16], SikftOS [17], and Pelikan [7]. Notably, with the integration of SIEVE, Ristretto got a 33x reduction in cache misses and 16% CPU savings.

Theodon: A Modular Framework for CDN Optimization

A CDN is a large, globally distributed system comprising hundreds of thousands of servers, playing a vital role in various internet services. However, the complexity of CDN operations presents substantial challenges. Despite their widespread use, pinpointing an optimal configuration that effectively balances key objectives like latency, throughput, and cost is a complicated endeavor.

Our system, THEODON, recommends configurations that balance the trade-off between performance and cost. It achieves this through modular simulation of CDN topologies, coupled with the learning of near-optimal configurations via multi-objective Bayesian Optimization. THEODON breaks down a CDN topology into interconnected components, each symbolizing a core aspect of the CDN infrastructure. By processing real CDN workloads, THEODON identifies configurations that adeptly balance performance with cost efficiency in CDN contexts.

Through simulating two real-world CDN systems, WikimediaCDN and Cloudflare, THEODON uncovers up to 10% and 23% reduction in terms of byte miss ratio, respectively. Furthermore, THEODON demonstrates the capability to discover configurations that substantially reduce costs – by approximately 2.4× – in Cloudflare, all while maintaining performance levels comparable to the default settings.

1.2 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 provides relevant background and motivation. Chapter 3 describes LatenSeer, a data-driven modeling framework for estimating end-to-end latency distributions in microservice-based web applications. In Chapter 4, we describe SIEVE, an algorithm that is simpler than LRU and provides better than state-of-the-art efficiency and scalability for web cache workloads. In Chapter 5, we present THEODON, a framework that models CDN architectures through modular simulations, aiming to recommend configurations that adeptly balance performance with cost. We conclude with a summary of our main contributions and a discussion of future work directions in Chapter 6.

Chapter 2

Background

This chapter introduces three representative complex distributed systems involved in this thesis: microservice-based applications, large-scale web cache systems, and CDNs. We discuss the inherent complexities and the need for performance modeling in these systems. Additionally, we explore the opportunities and challenges of leveraging empirical data for performance modeling.

2.1 Microservice-based Applications

2.1.1 Complicated Dependencies and Interactions

The microservice architectural style involves developing a single application as a suite of small, independently functioning services. Each of these services operates in its own process and communicates using lightweight mechanisms, often through an HTTP resource API. Many large-scale applications comprise hundreds of geo-distributed, loosely coupled microservices that exhibit complex inter-dependencies [90]. Such an architecture has many benefits, including rapid microservice evolution, and the ability to use flexible microservice placement to trade off performance and cost. On the other hand, the proliferation of microservices adds complexity: each microservice may contact any number of other microservices on behalf of the user-facing request being served, with intermediate microservices potentially creating a cascade of serial or parallel calls to other components.

2.1.2 The Need for Modeling Latency

Software systems are constantly evolving; these changes often have implications on latency. Tech giants such as Amazon, Google, and Netflix implement thousands of daily deployments and production changes across hundreds of services that comprise their production environments [84, 118, 173]. Estimating the impact of these frequent production changes ahead of time is crucial, especially for managing risk, ensuring system stability, and maintaining Service Level Objectives (SLOs).

Latency estimation is the task of predicting the impact of hypothetical service changes on end-to-end latency—the entire execution latency of requests to some top-level API endpoint across all the services involved in its execution. The operators of large-scale services are accountable for balancing the service response times with features, resource utilization, policy constraints, and costs. When evaluating potential changes to a service, the operators face the conundrum of assessing how the modifications might impact the response time—a critical factor in end-user experience.

To further elucidate, we highlight the importance of latency estimation by considering several "high-stakes" service changes.

Adding a new microservice. When a new feature is introduced into the application, one or more new microservices may be added to the system. While the new service may enhance the customer experience, it might lengthen response times as requests to the platform must now pass through the additional microservice.

Scaling strategies. An operator of a video streaming service might consider scaling up their infrastructure to handle more concurrent users. While this might increase the ability to handle high traffic, it could also introduce additional latency due to the complexity of interactions among servers or data replication across multiple locations.

Policy changes. In some cases, deployment changes may be required due to policy constraints. For instance, new regulations might require user data to be stored only in certain geographical locations. This could potentially extend the end-to-end latency due to increased data transfer times. Latency estimation could help the operator understand the impact of these changes and plan mitigation accordingly.

Latency estimation questions are *interventionist*: they concern hypothesizing how a running system will behave after it is changed. Conventional statistical and machine learning tools, relying on observational data, fall short in answering these queries due to their associative nature [160] and susceptibility to confounding variables [159]. Moreover, while randomized controlled trials offer more precise results, they are resource-heavy and inflexible [85, 186].

2.1.3 The Proliferation of Distributed Tracing

Distributed tracing confers a great opportunity for piggybacking latency estimation on existing data, providing end-to-end visibility, and revealing service dependencies. Distributed tracing systems have been used in microservices or serverless applications to track the performance. Many production systems deploy distributed tracing frameworks, such as OpenTracing [152], Zipkin [244], and Jaeger [102], to track request traces and aggregated metrics, which is useful to examine hand-off between system components [32, 156, 177], and to troubleshoot practical system problems such as slow responses and errors [26, 77, 83, 87, 108, 124, 131, 151, 197]. The primary use cases of distributed end-to-end tracing systems concern active system monitoring [32, 156, 177], anomaly detection [26, 52, 87, 115, 130, 167, 174, 178], and root-cause analysis [153, 213].

In our context, trace aggregation can be used to delineate the interdependencies among all services within a microservices system—allowing for data that is routinely

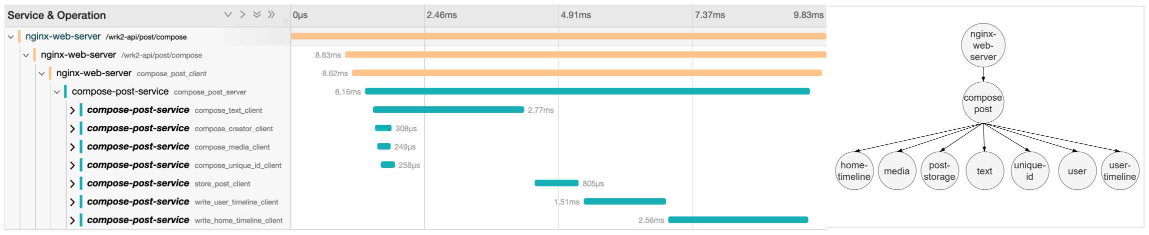

Service & Operation
0µs
2.46ms
4.91ms
7.37ms
9.83ms
nginx-web-server /wrk2-api/post/compose
nginx-web-server /wrk2-api/post/compose
8.83ms
nginx-web-server compose_post_client
8.62ms
compose-post-service compose_post_server
8.16ms
compose-post-service compose_text_client
2.77ms
compose-post-service compose_creator_client
308µs
compose-post-service compose_media_client
249µs
compose-post-service compose_unique_id_client
258µs
compose-post-service store_post_client
805µs
compose-post-service write_user_timeline_client
1.51ms
compose-post-service write_home_timeline_client
2.56ms
nginx-web-server
compose post
home-timeline
media
post-storage
text
unique-id
user
user-timeline

Figure 2.1: An example trace from Jaeger. The left diagram shows the trace in a timeline view, and the right diagram shows the service dependency graph generated by Jaeger.

collected to be useful for latency estimation **(G1)**. The left side of Fig. 2.1 shows a visual example of a trace constructed by Jaeger [102], a state-of-the-art open source tracing framework, for a compose post request to a benchmark of social network application [86]. An end-to-end *trace* represents an execution path through the system, whereas a *span* represents a logical operation from a function. A span maintains not only the causal relationship by keeping a reference to a parent span but also the runtime execution details, such as start and end timestamps to represent the duration of the operation. A trace can be considered as a directed acyclic graph (DAG) of spans, where each edge represents the causal relationships, such as RPC calls, between two spans [177]. Besides tracking causal relationships, a trace also captures temporal order between a group of child spans that are concurrently called by the same span.

We show an illustration of a service dependency graph, created from 20 traces of `compose-post` requests, in the right-hand-side diagram of Fig. 2.1. The completion of service `compose-post` is contingent on even other services, such as service `media` and service `text`. Below, we refer to the services that are invoked by the same "parent" service in a service dependency graph as *sibling nodes*.

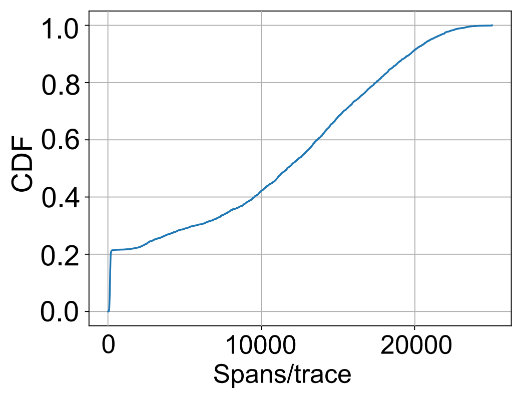
2.1.4 Challenges of Using Distributed Traces to Estimate Latency

Distributed tracing data cannot provide end-to-end latency estimation out of the box due to multiple challenges.

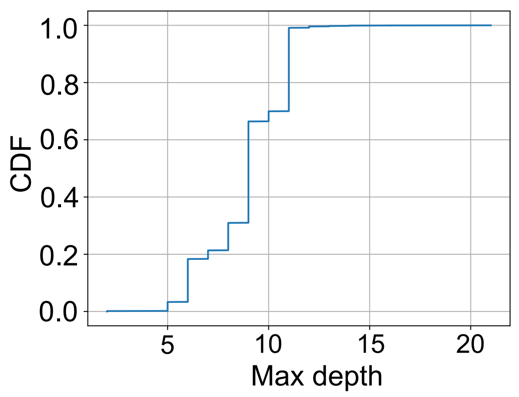
(1) Raw traces fail to capture interdependencies. The perspective offered by each individual trace is too insular to fully encapsulate the complexity of the entire system. Yet aggregation techniques to produce service dependency graphs exhibit significant blind spots. Chief among them is the problem that sibling nodes may themselves have complex interdependencies that, when uncaptured, can result in an incomplete portrayal of the performance dynamics and intricacies, particularly in hypothetical scenarios, thereby undermining the accuracy of latency estimation. Nevertheless, a cautious and careful approach can overcome these limitations, as shown in §3.2, allowing distributed tracing to be valuable resource for latency estimation.

(2) Small inaccuracies can snowball. The inevitability of clock skew is a major issue for nuanced processing of distributed tracing data. In a distributed tracing deployment, spans are timestamped using the local machine's clock upon start and finish. However, the clocks on different machines in a distributed system invariably drift apart, even with periodic synchronization using the Network Time Protocol (NTP) [112, 122, 141, 149, 175]. This well-understood problem in distributed computer stems from various factors, such as clock hardware differences, environmental conditions, network latency, and the resolution of the system clock. Consequently, this can lead to misinterpretations in system performance analysis and incorrect event ordering [53, 235], with minor errors potentially amplifying to significantly impact conclusions on performance behavior.

(3) The problem of scale. A single trace is not representative of the full service—any interventional questions must account for the diversity of traces that execute in a production environment. At Twitter, for instance, a request involves 12,000 spans on average, with some traces encompassing as many as 25,000 (Fig. 2.2a). The call graphs for one endpoint comprise between 1 and 25,000 spans at tree depths between 2 to 22 (Fig. 2.2b) and encompass widths between 1–5,000. Uber, similarly, operates nearly 4,000 microservices, and a single request trace can have up to more than 11,000 spans



(a) Median spans/trace=11676



(b) Median max call path depth=9

Figure 2.2: Twitter trace characteristics.

nested 40 levels deep [235]. Existing tracing tools focus on providing the operator with a single, specific trace [98, 131], such as to identify an edge-case in Jaeger [230], or analyzing the path of a single request [131]. How to properly aggregate thousands, or even millions, of distributed traces to gather insights is a nascent and understudied area [235].

In Chapter 3, we introduce LatenSeer, a framework that models the complex relationships between services by leveraging distributed traces. It practically and accurately models end-to-end latency at scale.

2.2 Web Caching Systems

Web caches are essential components of modern Internet infrastructure, playing a crucial role in reducing data access latency and network bandwidth. Key-value caches, e.g., Memcached [5], Pelikan [7] and Cachelib [63], are widely used in modern web services such as Twitter [225] and Meta [38] to reduce service latency. CDN caches are deployed close to users to reduce data access latency and high WAN bandwidth cost [28, 216, 223, 234].

2.2.1 Increasing Complexity in Cache Eviction Policies

The cache eviction algorithm, which decides which objects to store in the limited cache space, governs the performance and efficiency of a cache. The field of cache eviction algorithms has a rich literature [25, 34–36, 41, 48, 55, 59, 62, 65, 69, 72, 73, 78, 96, 103, 119, 120, 155, 168, 181, 192, 205, 211, 215, 237].

Increasing complexity. Most works on cache eviction algorithms focused on improving efficiency, such as LRU-k [150], TwoQ [106], SLRU [110], GDSF [48], EELRU [180], LRFU [65], LIRS [104], ARC [139], MQ [242], CAR [30], CLOCK-pro [103], TinyLFU [70, 71], LHD [33], LeCaR [202], LRB [182], CACHEUS [170], GLCache [219], and HALP [183]. Over the years, new cache eviction algorithms have gradually convoluted. Algorithms from the 1990s use two or more static LRU queues or use different recency metrics; algorithms from the 2000s employ size-adaptive LRU queues or use more complicated recency/frequency metrics, and algorithms from the 2010s and 2020s start to use machine learning to select eviction candidates. Each decade brought greater complexity to cache eviction algorithms. Nevertheless, as we show in §4.3, while the new algorithms excel on a few specific traces, they do not show a significant improvement (and some are even worse) compared to the traditional ones on a large number of workloads. The combination of limited improvement and high complexity explains why these algorithms have not been used in production systems.

The trouble with complexity. Multiple problems come with increasing complexity. First, complex cache eviction algorithms are difficult to debug due to their intricate logic. For example, we find two open-source cache simulators used in previous works have two different bugs in the LIRS [104] implementation. Second, complexity may affect efficiency in surprising ways. For example, previous work reports that both LIRS and ARC exhibit Belady's anomaly [91, 203]: miss ratio increases with the cache size for some workloads. It's worth noting that FIFO, although simple, also suffers from this anomaly. Third, complexity often negatively correlates with throughput performance. A more intricate algorithm performs more computation with potentially longer critical sections, reducing both throughput and scalability. Furthermore, many of these algorithms need to store more per-object metadata, which reduces the effective cache size that can be used for caching data. For example, the per-object metadata required by CACHEUS is 3.3× larger than that of LRU. Fourth, complex algorithms often have parameters that can be difficult to tune. For example, all the machine-learning-based algorithms include many parameters about learning. Although some algorithms do not have explicit parameters, e.g., LIRS, previous work shows that the implicit ghost queue size can impact the efficiency [203].

Simple eviction algorithms. Besides works focusing on improving cache efficiency, several other works have improved cache throughput and scalability. For example, MemC3 [79] uses Cuckoo hashing and CLOCK eviction to improve Memcached's throughput and scalability; MICA [125] uses log-structured storage, data partitioning, and a lossy hash table to improve key-value cache throughput and scalability. Segcache [226] uses segment-structured storage with a FIFO-based eviction algorithm and leverages macro management to improve scalability. Frozenhot [162] improves cache scalability by freezing hot objects in the cache to avoid locking. However, these works often use weaker eviction algorithms such as CLOCK[1] and FIFO.

2.2.2 Open-sourced Cache Workloads

Historically, evaluations of caching systems were often constrained, typically relying on just one or two traces for analysis. While this approach yielded some insights, it was notably limited. A significant advancement has been the open-sourcing of a diverse range of caching workloads. This development includes a collection of web cache and block traces sourced from real-world production environments, providing a much broader and more informative basis for evaluating caching systems:

[1]CLOCK was recently shown to be more efficient than LRU [220].

- Web cache traces: examples include Twitter cache cluster workloads [225], Wikimedia CDN traces [208], Meta key-value/CDN traces [9], and Tencent photo traces[239, 240].
- Block traces: these include workloads from MSR [142, 143], FIU [113], Cloudphysics [204], and Alibaba [1, 121, 206].

Web cache workloads typically follow Power-law (generalized Zipfian) distributions [38, 45, 46, 58, 89, 94, 99, 187, 191, 225], where a small subset of objects account for a large proportion of requests. In detail, the i^{th} popular object has a relative frequency of $1/i^{\alpha}$, where α is a parameter that decides the skewness of the workload. Previous works find different α values from 0.6 to 0.8 [45], 0.56 [89], 0.71–0.76 [93], 0.55–0.9 [38], and 0.6–1.5 [225]. The reasons for the large range of α include (1) the different types of workloads, such as web proxy and in-memory key-value cache workloads; (2) the layer of the cache, noting that many proxy/CDN caches are secondary or tertiary cache layers [99]; and (3) the popularity of the service, such as the most popular objects receiving greater volume of requests in more popular (widely-used) web applications. Moreover, web caches often serve constantly growing datasets — new content and objects are created every second.

In contrast, the backend of enterprise storage caches or single-node caches, such as the page cache, often has a fixed size, not regularly observing new objects. Further, many storage cache workloads often have scan and loop patterns [170], in which a range of block addresses are sequentially requested in a short time. Such patterns are rare in web cache workloads according to our observation on 1559 traces from 7 datasets.

2.2.3 Opportunity: Lazy Promotion and Quick Demotion

Promotion and demotion are two cache internal operations used to maintain the logical ordering between objects[2]. Recent work [220] shows that "lazy promotion" and "quick demotion" are two important properties of efficient cache eviction algorithms.

Lazy promotion refers to the strategy of promoting cached objects only at eviction time. It aims to retain popular objects with minimal effort. An example of lazy promotion is adding reinsertion to FIFO. In contrast, FIFO has no promotion, and LRU performs eager promotion – moving objects to the head of the queue on every cache hit. Lazy promotion can improve (1) throughput due to less computation and (2) efficiency due to more information about an object at eviction.

Quick demotion removes most objects quickly after they are inserted. Many previous works have discussed this idea in the context of evicting pages from a scan [33, 106, 139, 150, 170, 180]. Recent work also shows that not only storage workloads but web cache workloads also benefit from quick demotion [220] because object popularity follows a power-law distribution, and many objects are unpopular.

Given these insights, especially in the context of web cache workloads, the concepts of lazy promotion and quick demotion inspire a reevaluation of cache eviction algorithms. In Chapter 4, we present Sieve, a simpler eviction algorithm that achieves better than state-of-the-art efficiency and scalability for web cache workloads.

2.3 Content Delivery Networks

A Content Delivery Network (CDN) is a network of cache servers distributed geographically, designed to deliver web content and services to users more rapidly and efficiently [132, 245]. For example, a CDN may reduce the latency of content delivery

[2]Note that the terms "promotion" and "demotion" are also commonly used in the context of cache hierarchy. In this case, promotion refers to the process of moving data to a faster device, while demotion involves moving the data to a slower device [136, 210].

from 30 milliseconds to 150 milliseconds [99, 137, 148, 241].

A typical CDN consists of a hierarchical arrangement of servers: back-end servers manage internal content distribution within the CDN, while front-end servers at the network edges handle user interactions. CDN providers use complex algorithms to route user requests to the most appropriate server [132]. The journey of a request typically starts from the end-user, proceeding to the chosen front-end server. If this server already has the requested content, it is immediately delivered to the user. Otherwise, the server forwards the request up the CDN hierarchy until it finds the content, which might sometimes involve retrieving it from the content provider's original server.

2.3.1 CDN Architectures

Content providers select CDN strategies based on their scale and needs, creating a convoluted landscape of CDN use. Smaller providers with limited budgets often use cost-effective options like centralized hosting or free CDNs. Larger commercial entities prefer customized services from major CDNs like Akamai [148] for competitive delivery performance. The biggest providers, such as Netflix [18] and Youtube [185], develop their own specialized CDNs due to their extensive requirements. This diversity of requirements has led to the CDN ecosystem becoming increasingly complex, featuring a wide variety of CDN architectures. In the following, we outline some representative examples of these architectures.

- **Datacenter-based CDNs:** Some CDN operators concentrate large numbers of servers in a relatively small number of geographic locations. such as fastly [209] and Cloudflare [15].
- **Highly Distributed CDNs:** Akamai is the leading CDN provider with a highly distributed server network. By strategically placing edge servers in closer

proximity to end-users – in terms of both network and geographical distance – and utilizing optimized protocols, Akamai's distributed CDN architecture achieves significant performance enhancements [132].

- **Specialized CDNs:** Several large content providers, which previously depended on third-party CDN services, have shifted towards developing their own application-specific CDN platforms, such as WikimediaCDN [169] and Youtube [185].

The complexity in the CDN ecosystem is not just confined to the variety of architectures but also extends to the intricate design decisions involved in content storage and retention. Factors such as which servers will store specific content, the length of time for content retention, load balancing, caching eviction algorithms, and admission control, are all critical in determining a CDN's operational efficiency and performance.

2.3.2 Cost of CDN

The primary business model of CDNs revolves around generating profits by balancing the revenue received from users against their operational costs. Typical costs for a CDN include expenses for bandwidth used in packet transit, acquiring and maintaining servers (which also encompasses the energy required to power and cool these servers), and the costs associated with CDN personnel. When CDNs first emerged in the late 1990s, bandwidth prices were relatively high, constituting a significant portion of the total content delivery costs. In recent times, despite the general global trend of decreasing unit prices for bandwidth, substantial regional price variations remain [190]. For instance, bandwidth costs in Asia and the Pacific can be up to three times higher than those in Europe and North America [189]. Consequently, minimizing bandwidth costs has been a key focus for most CDN providers. Additionally, there has been a shift towards using more SSDs in CDN operations due to their lower costs compared to DRAM [38, 99, 185].

2.3.3 The Need for Performance and Cost Modeling

Considering the previous discussion, CDNs possess inherent complexities in their architecture and costs, which align with the evolving requirements of the market. A comprehensive approach is required to determine the most effective strategy. The performance of a CDN hinges not only on its architectural design but also on various factors such as the nature of the content (static vs. dynamic) and network conditions. To navigate these complexities, empirical data is essential for guiding decisions like where to establish new data centers, determining the required capacity and hardware types for cache servers, and more.

Due to the expansive scope of CDN architectures, this thesis primarily focuses on the architectural design of data centers. Future work will delve into the placement and hierarchical design between data centers. We take into account the cost of servers, and the bandwidth cost is inferred from the byte miss ratio at a Point of Presence (PoP), which indicates the amount of traffic offloaded to the origin. In Chapter 5, we present Theodon, a framework designed to model CDN architectures using modular simulations. Theodon's objective is to identify near-optimal configurations that effectively balance performance with cost considerations, providing a strategic approach to CDN management and optimization.

Chapter 3

LatenSeer: Causal Modeling of End-to-End Latency Distributions by Harnessing Distributed Tracing

3.1 Introduction

Latency estimation enables operators managing large-scale web applications [53, 105, 123, 134] to anticipate the impacts of potential changes before scaling their services, optimizing costs, adding features or adapting to changes in hardware configuration or cloud computing deployments [105, 126, 130, 213]. For instance, the operators may wish to assess how introducing a novel machine learning pipeline or migrating some subset of services to a different data center would affect customer-perceived latency. Such estimation is crucial to prevent performance regression and to protect the end-user experience: Google, Amazon, and Akamai have all noted significant drops in traffic or revenue following a modest (100+ ms) increase in latency [19, 81, 114, 195].

To be pragmatic, operators of web applications desire a latency estimation framework that meets the following goals.

(G1) **Easy deployment.** The system integrates with existing data collection systems, eschewing invasive and labor-intensive custom instrumentation.

(G2) **Flexible scenarios.** The system estimates end-to-end latency given hypothetical latency changes in any of its constituent services, informing advanced decision-making.

(G3) **Realistic forecasts.** The system should express confidence in its results—sound decisions require sound data and models.

In large-scale systems, however, these goals are challenging to fulfill. At scale, modern applications often comprise hundreds or thousands of loosely coupled microservices [57, 90, 128, 176, 196, 238], where a single user request may touch thousands of instances before generating a response [154, 176, 235]. In microservices architectures, a single service often interacts with many others, forming a complex web of serial or parallel calls to fulfill a user request. Modifications in one can thus affect others, leading to varied impacts on the end-to-end latency of different requests.

State-of-the-art solutions fall short of meeting all three objectives. Several works [105, 165, 166] require explicit instrumentation that makes them difficult and expensive to deploy in different environments (**G1**). Other methods, which use purely data-driven techniques, either do not target microservice-based web applications [95, 134, 193, 201], or are restricted to making predictions for a subset of services [68, 235] (**G2**). Finally, the trend of leveraging deep machine learning models for latency estimation would attempt to draw causal conclusions through black-box methods that fundamentally can derive only associations [158, 159], and are thus set up to fail goal (**G3**) [172].

We present LatenSeer, a framework for estimating the end-to-end latency distribution for large-scale microservices applications, that is explicitly designed to fulfill goals **(G1)—(G3)**. As shown in the upper left of Fig. 3.1, LatenSeer piggybacks on the proliferation of distributed end-to-end tracing in large-scale systems, such as

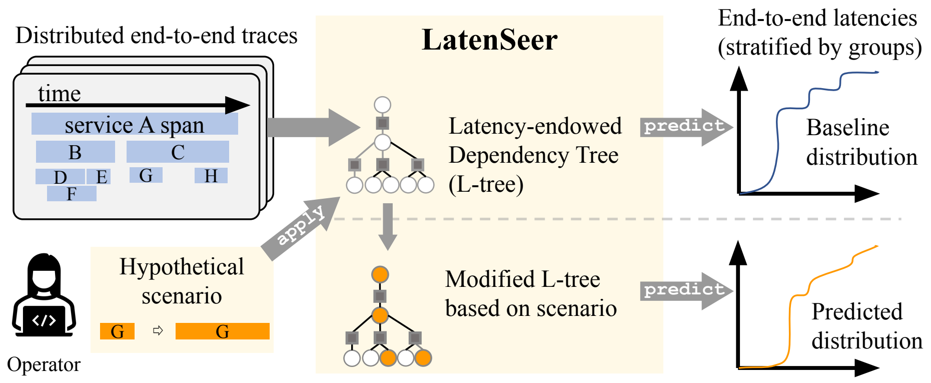

Distributed end-to-end traces
time
service A span
B
C
D
E
F
G
H
LatenSeer
Latency-endowed Dependency Tree (L-tree)
predict
End-to-end latencies (stratified by groups)
Baseline distribution
apply
Operator
Hypothetical scenario
G
G
Modified L-tree based on scenario
predict
Predicted distribution

Figure 3.1: Overview of LatenSeer. Operators can pose hypothetical scenarios into a trace-driven causal model and predict how end-user latency would change relative to baseline.

Jaeger [102] or OpenTelemetry [151], rather than demanding custom instrumentation **(G1)**. LatenSeer allows an operator (bottom left) to answer the question: *How will the end-users be impacted by some hypothetical changes to the latencies of the underlying microservices* **(G2)**? LatenSeer accepts hypothetical latency changes to services as inputs, and outputs the changed end-to-end latency distribution. We apply LatenSeer on two use cases of latency estimation: *service placement* (UC1)—reasoning about the end user latency impact of resource provisioning or service migration, and *slack analysis* (UC2)—determining the latency budget available to alter a microservice without impacting the overall response times of requests. We validate LatenSeer via null prediction [1] on distributed traces from two production sources to show that the underlying model is sound, and conduct controlled experiments using a social network microservices-based benchmark to evaluate the accuracy of latency estimation **(G3)**.

Inside LatenSeer, we cast the latency estimation problem as a causal model [2] of latency components through a simple principle: a complex service decomposes into the causal interactions between its constituent microservices. The end-to-end latency distribution for the entire application is thus the combination of each component's

[1]Null prediction refers to the experiments where LatenSeer derives the predicted latency distribution under conditions of zero injected delay.

[2]In distributed tracing, "causality" usually refers to the order of events or operations as they happened in relation to each other

latency distributions.

LatenSeer overcomes multiple technical challenges that arise when forecasting latency in large-scale systems. First, distributed traces, which record the requests within distributed applications, often exhibit diverse execution paths. Moreover, a single service might be involved in multiple top-level APIs. As a result, aggregating the distributed traces requires a data structure that is simultaneously succinct while maintaining sufficient diversity of requests to ensure accurate latency estimation [116]. A key idea in LatenSeer is the use of *set nodes* to efficiently cluster traces that exhibit similar execution paths.

Second, the latency of a parent service is a composite of its child services' latencies, underscoring the importance of discerning all causal relationships. Identifying causal dependencies from single requests is straightforward but aggregating them is complex due to clock skews and path-dependent executions. To address this, LatenSeer introduces the *succession time* alongside *set nodes* to aid in deducing the causal dependencies in child RPCs demonstrating similar execution execution paths

Finally, to improve the accuracy of latency estimation, LatenSeer profiles the joint distribution of child service latency, departing from the convenient but misleading assumption of mutual independence upon which most traditional approaches have relied. Joint latency profiling accounts for scenarios where the latency of different child services may be interrelated, thus making latency estimates more accurate than what previous models could achieve.

We implemented LatenSeer as a Python package, and use it in a social network prototype built from the DeathStarBench microservices benchmark (DSB) [86]. We evaluate LatenSeer's ability to make accurate latency predictions through two real world scenarios, service placement (UC1) and slack analysis (UC2). Our results show that LatenSeer predicts end-to-end latency distributions within a 5.35% error (D-statistic) with a mean of 2.7%. In contrast, the state-of-the-art gives a minimum error surpassing 9.5% with a mean of 14.5%. Finally, we evaluate LatenSeer using two production traces (Alibaba [128] and Twitter). We verify the soundness of latency prediction through null prediction experiments and demonstrate the scalability of LatenSeer on massive production workloads.

In summary, this chapter makes the following contributions.

- We propose trace-driven causal modeling as a methodology for answering interventional latency estimation questions in large microservices architectures.
- We detail the design of the LatenSeer framework for extracting causal interservice dependencies from traces generated by off-the-shelf distributed tracing systems to generate causal models of end-to-end latency.
- We demonstrate LatenSeer's utility by accurately answering hypothetical questions on realistic scenarios with an estimation error within 5.35% (D-statistic), outperforming the state-of-the-art.
- We show LatenSeer's scalability through experiments on real-world traces from large-scale, complex production systems.

3.2 Design and Implementation

LatenSeer is an offline tool for conducting latency estimation from distributed traces. This tool permits the system operators to infer how end-to-end latency would be affected when the latencies of specific services are changed. In this section, we describe how LatenSeer overcomes the aforementioned challenges to derive a causal model of steady-state latency from historic distributed trace data of the system.

Interface. Fig. 3.2 shows LatenSeer in action, estimating end-user latency distributions under the assumption that load-balancing services have been migrated to a distant data center (UC1). The operator first obtains a baseline model of latencies based on trace data. As input, the operator poses an interventional query to the model by defining a subset of services to move (*e.g.*, '`LBS`'), and the delay incurred (or reduced) from the movement (*e.g.*, a normal distribution centered on 30 ms) triggering the latency propagation via `model.apply` which returns a latency-modified model, obtaining a predicted latency distribution as output (via `scenario.predict`). Separately, the operator also determines the available latency budget for the `LBS` service through a `model.slack` call, which returns CDFs of latency slacks for all services (UC2). The latency distributions in LatenSeer can be further stratified by arbitrary groups (*e.g.*, to assess latency impacts on customers in certain regions)—we focus on a single group for the clarity of presentation.

Key properties. To estimate end-to-end latency under potential alterations in any component services, LatenSeer constructs a model that fulfills the following properties: (1) it delineates the causal dependencies between microservices. (2) it recognizes the range of request routes within the system, accounting for the different paths a request might take and the specific services it might encounter. (3) it maintains the latency distributions of interactions (e.g., RPC or REST) between each pair of communicating services.

```
# Load distributed trace data
model = LatenSeer(data="s3://trace_bucket")
# UC1:Add hypothetical 30ms delay to service 'LBS'
scenario = model.apply('LBS'=lambda x:x+30)
# Get the changed end-to-end latency distribution
E2E = scenario.predict()
# UC2:Determine slack of 'LBS' service
LBSslack = model.slack('LBS')
```

Figure 3.2: LatenSeer Example. Estimate latency distribution if load balancing services are migrated to another data center with 30ms extra delay.

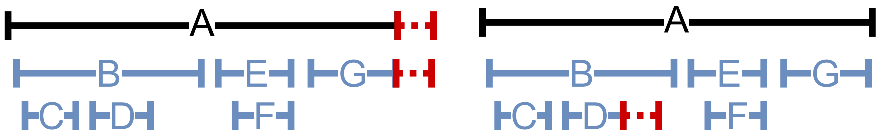


Figure 3.3: Child service latency may affect parent service.

3.2.1 Modeling Latency with Invocation Graph

We begin by elucidating the principles behind latency calculation using a single trace as an illustrative example.

The underlying concept is simple: the latency of a parent service can be deduced from the latencies of its child services. For example, Fig. 3.3 illustrates a trace consisting of a parent service A and six child services (B-G), where B, E, and G are on the latency-critical path. We define the following functions on a particular trace: $\mathcal{L}(x)$ the latency of service x, and $\mathcal{L}(\mathcal{N})$ which accounts for the network latency or other processing time on the latency-critical path. For instance, it includes the duration between B's finish time and E's start time. Consequently, the latency of the span A can be expressed as $\mathcal{L}(A) = \mathcal{L}(B) + \mathcal{L}(E) + \mathcal{L}(G) + \mathcal{L}(\mathcal{N})$.

Now we discuss how latency changes on different child services impact the latency of the parent service. Since services B, E, and G are on the latency-critical path, any delay in these services will increase the latency of A. However, the latency of A can also be affected by other services that are off the latency-critical path, such as C, D, and F in this example. We summarize four scenarios of latency change on a service and their impacts[3]:

1. An increase in the latency of a child on the latency-critical path invariably leads to an increase in the parent latency
2. A decrease in the latency of a child on the latency-critical path can result in unchanged or reduced parent latency, potentially altering the latency-critical path itself.
3. When a child off the latency-critical path experiences increased latency, the parent latency might remain the same or increase, contingent upon changes to the latency-critical path.
4. A decrease in the latency of a child off the latency-critical path leaves the parent latency unaffected.

Fig. 3.3 provides two examples demonstrating cases (1) and (3) respectively.

As previously discussed, the latency-critical path exhibits dynamism due to latency changes in different services. To assess the impact of these changes on the latency of a parent service, we construct an ***invocation graph*** for the child services. This graph captures the sequential execution order of the child services within a specific trace. In the invocation graph, an edge $\mathcal{E}(x, y)$ represents service y finishes before service x happens. Node *start*, *end*, and *sync* are virtual nodes that denote the starting, finishing, and synchronization points, respectively. For example, node G will not start until both nodes E and F have finished. Fig. 3.4 shows a comparison between a standard service dependency tree and an invocation graph derived from the same trace (Fig. 3.3). Invocation graphs represents child relationships with finer granularity, a detail not captured by service dependency trees. Moreover, each node is endowed with a latency value corresponding to its service. The latency of the parent service

[3]We can effectively calculate the latency changes on multiple services. For ease of presentation, we only show latency change on one service.

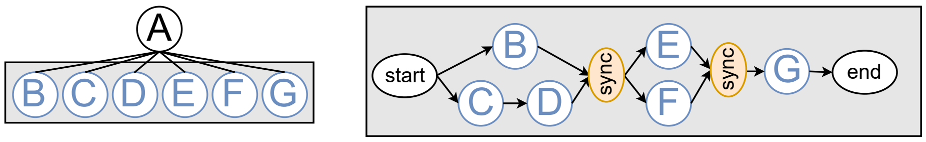


Figure 3.4: From service dependency tree to invocation graph.

can be expressed as $\mathcal{L}(A) = max(\mathcal{L}(B), \mathcal{L}(C) + \mathcal{L}(D)) + max(\mathcal{L}(E), \mathcal{L}(F)) + \mathcal{L}(G)$. Given this formulation, any alterations in a child service's latency allow for a direct computation of the resulting impact on the parent service's latency.

We now describe how the conventional service dependency tree can be converted to an invocation graph. We use *serial* and *parallel* to describe the relationship between any two spans. Consider, for example, spans B and E depicted in Fig. 3.3, we define the *succession time* as the difference between the starting time of E and the finishing time of B. Given that B starts before E from the vantage point of the parent observer, we say that B happens before E. If the succession time is positive, we declare the two spans to be serial. Conversely, a negative succession time, such as the overlapping time frames of E and F, indicates a parallel relationship between two spans.

The initial step involves identifying sync points, which serve to demarcate nodes into chronological groups. Each group is characterized such that services from a preceding group must complete their execution before the services in the succeeding group initiate. To establish these groups, we construct a graph $\mathcal{G}_p$ comprising child nodes and connect nodes that are in parallel relationships. Subsequently, we identify the connected components in the graph. Illustrating with the child nodes (B-G) in Fig. 3.4, we can construct a graph $\mathcal{G}_p$ consisting of nodes B, C, D, E, F, and G, interconnected through edges B-C, B-D, and E-F. Consequently, the connected components are $\{B, C, D\}$, $\{E, F\}$, and $\{G\}$. To complete this step, we arrange these groups chronologically based on the earliest start time among the nodes within each group.

The next step is to identify the causal orders among nodes within each group.

Within each group, we establish connection between pairs of child nodes that are in serial relationship, forming a graph denoted as $\mathcal{G}_s$. In this graph, the maximal cliques, or *independent sets* in graph theory parlance [88] are the nodes with serial relationships. And the nodes within each maximal clique are then ordered by their starting timestamps. For example, in the first group $\{B, C, D\}$, maximal cliques are $\{B\}$ and $\{C, D\}$.

Drawing upon the aforementioned two steps, we can determine the invocation order of the child nodes. When a latency change occurs in any given service, we can identify the slowest path within the invocation graph, thus determining the overall latency of the parent service.

3.2.2 Aggregating Traces with L-tree

We have presented the foundational principles of latency modeling. In this section, we detail the construction of a latency-endowed service dependency tree, referred to as the **L-tree**, which aims to refine aggregate-level latency calculations.

The construction of the L-TREE is a top-down process by two main steps. Firstly, we iterate through traces to establish set nodes between parent spans and their direct children, merging identical spans where necessary. This forms an initial L-tree structured by set nodes, albeit without invocation graphs. Secondly, we traverse this preliminary tree to construct invocation graphs at each set node, drawing upon stored latency profiles. This produces a detailed L-tree. The subsequent parts of this section will delve deeper into each stage of this construction process.

Set node. The *set node* functions to effectively cluster together traces that exhibit similar invocation graphs. Each *regular node* in the L-TREE therefore denotes a span in the original traces. When multiple traces include the same span, the corresponding node in the L-TREE represents aggregate information of that particular span across the traces of the same call path. Each regular node records how many traces had the

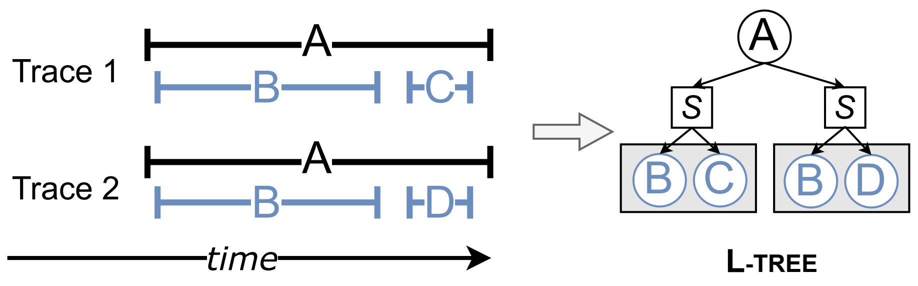


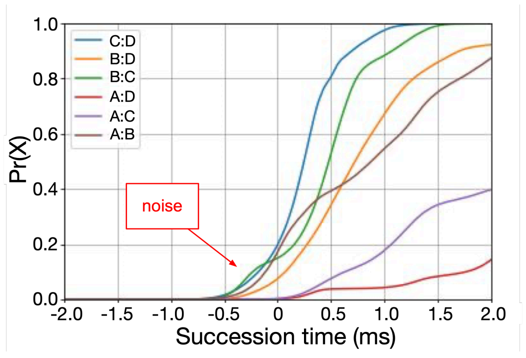
Figure 3.5: Aggregate traces to L-TREE.

span to which it corresponds. A set node connects a regular node with a collection of regular child nodes and indicates an identical set of child nodes—the same set of spans[4]. We iteratively input traces, and merge the new input trace into L-TREE; if a new set of child spans occur, we create a new set node. Moreover, each set node remains a record of the number of traces it encompasses, thereby providing insights into the different branch ratios present in the L-TREE. For instance, Trace 1 and Trace 2 in Fig. 3.5 have same parent span A, but different sets of child spans $\{B$, $C\}$ and $\{B$, $D\}$. The right diagram in Fig. 3.5 shows a L-TREE for these two traces, where two set nodes are created to connect the parent node A and two different sets of child nodes.

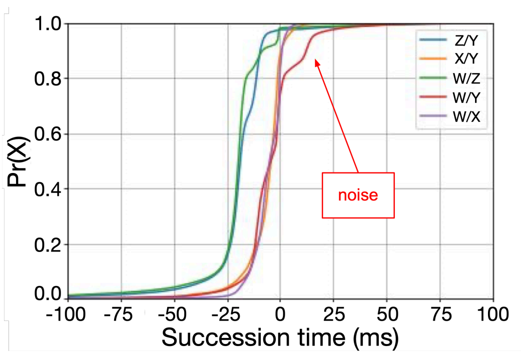
Set nodes can cluster traces that exhibit similar invocation graphs, which effectively retain the diverse characteristics of calling relationships, thereby enhancing the modeling of latency distribution. In constructing the tree from the top down, traces sharing the same set of child RPC calls from a single parent call are clustered at each level. This approach strikes a balance between coarse aggregation (grouping all spans without differentiation) and fine-grained aggregation (grouping traces only if they share identical invocation graphs).

Joint latency profiling. Traditional service dependency modeling methods only store latency for each node independently [105, 134], which misses the finer-granular causalities between child services. L-TREE maintains latency profiles differently, by

[4]A set node effectively represents a *hyperedge* in a tree-based hypergraph.



(a) Known serial siblings.



(b) Known parallel siblings.

Figure 3.6: Succession time CDF for production services at Twitter over a 24-hour period.

jointly considering the latencies of sibling nodes. When forming a tree from a single trace and extending it to multiple traces, the latency data associated with each node is profiled as a distribution, rather than a single value. The set node manages the latency profiles of its child nodes. Besides maintaining the latency distribution for each child node, we also profile the succession time distribution between pairwise sibling nodes, which is crucial to construct an invocation graph for the sibling nodes.

Constructing invocation graphs in L-tree. Finally, we describe the construction of invocation graph in L-TREE. This process resembles the one described in §3.2.1, but with a key modification: we use the median of the succession time distribution between two sibling nodes as a threshold to determine their serial and parallel relationships. We hypothesize that service invocations that occur in series should consistently have positive succession time between them, whereas parallel spans may show negative succession time in some traces. Unfortunately, in production settings, the measurements are not always clear-cut, as shown in Fig. 3.6, due to clock skew and other instrumentation artifacts. The adoption of the median value as the classification threshold serves to mitigate these challenges, promoting a more accurate delineation of serial and parallel relationships.

This concludes our construction of L-TREE, where each regular node links to one or more set nodes indicating different call paths. Each set node maintains an invocation graph between the child nodes of the set node's parent node. Consequently, L-TREE not only maintains similar call paths across different tree branches but also captures the sequential execution order of child services.

3.2.3 End-to-end Latency Estimation

LatenSeer estimates the end-to-end request time by combining the latency distributions in L-TREE, propagating them bottom-up from leaves to the root. If an operator thus modifies the latency of specific nodes within the L-TREE that corresponding to the services they tend to change, LatenSeer will infer the latency impact from these changes through the modified latency distribution of the root node. For example, service B is delayed by Δ_B in Fig. 3.5, LatenSeer propagates the increased latency Δ_B, calculates the changed latency Δ_A for service A, and outputs A's new latency distribution $\mathcal{L}(A)'$.

With the collection of traces, LatenSeer models node latency as probability distributions. Formally, for serial nodes with latency distributions represented as random variables X_1,... X_k, we define ADD operator to estimate their combined distribution as: $\mathbb{P}(Z = z) = \sum \mathbb{P}(X_1 = x_1, ..., X_k = x_k)$, where $z = \sum_{i=1}^{k} x_i$. On the other hand, we define MAX operator to estimate their combined latency: $\mathbb{P}(Z \leq z) = \mathbb{P}(X_1 \leq z, ..., X_k \leq z)$.

Node latency estimation. We estimate the latency distribution of node v through its set nodes and corresponding invocations graphs of their child nodes. Suppose that a set node s connects an invocation graph $\mathcal{G}$ containing m sync points. Assume there are p_j paths between two sync points and n_j nodes on j-th path. We use P_j^i to denote the set of nodes on j-th path between $(i-1)$th and ith sync points. Then the combined latency ℓ_i of nodes between $(i-1)$th and ith sync points can be calculated

Algorithm 1 Latency Estimation

```
Require: L-TREE $G$; a "what-if" scenario $S$
Ensure: Estimated end-to-end latency distribution
 1: function PREDICT($G$, $S$)
 2:     $r \leftarrow$ root node of $G$
 3:     $A \leftarrow$ APPLY($r$, $S$)
 4:     $d \leftarrow$ greatest depth among nodes in $A$
 5:     while $d > 0$ do
 6:         Nodes $\leftarrow A[d]$
 7:         for each regular parent $p$ of a node in Nodes do
 8:             $\ell$, affected $\leftarrow$ LATENCYPROPAGATION($p$, $A$)
 9:             if affected then
10:                 update distribution of $p$ with $\ell$
11:                 $A[p.\text{depth}].\text{insert}(p)$
12:         $d \leftarrow d - 1$
13:     return updated distribution of root node $r$
```

as:

$$\ell_i = \textsc{Max}(\textsc{Add}(P_1^i), ..., \textsc{Add}(P_{p_1}^i))$$

Finally, the latency of set node s is calculated as:

$$\mathcal{L}(s) = \textsc{Add}(\ell_1, ..., \ell_m)$$

Suppose the latency distribution of the node v depends on k set nodes, denoted as s_i. The latency $\mathcal{L}(v)$ can be expressed as a weighted sum of the latencies of these set nodes, represented as $\sum_{i=1}^{k} w_i \mathcal{L}(s_i)$, where $w_i \in [0, 1]$ denotes relative weights with $\sum_{i=1}^{k} w_i = 1$. The relative weight is determined based on the number of traces recorded by each set node.

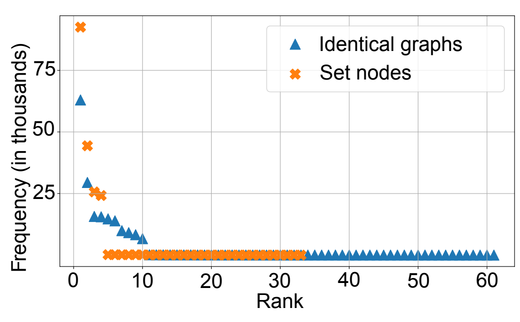
End-to-end latency estimation. LatenSeer combines the estimated latency distributions of all nodes in the L-TREE to produce one estimated distribution of end-to-end latency. We first inject the changed latency ($\Delta \in \mathbb{R}$) to the target nodes that correspond to the services for which the operator wants to intervene. (In causal modeling, this change emulates a do-operator [159, 160]). The algorithm works *bottom-up*, taking as input the L-TREE, and the set of hypothetical services to be altered with their corresponding latency (Alg. 1). The changes trigger bottom-up latency propagation within the L-TREE from the affected nodes toward the root. If a node displays variable latency, we recalibrate its parent node's latency using the ADD and MAX operators, following the structure of the invocation graph. Importantly, any modification in the latency of a parent node prompts a subsequent recalculation of its own parent, continuing this chain recursively up to the root node.

3.2.4 Making LatenSeer Practical for Production Workloads

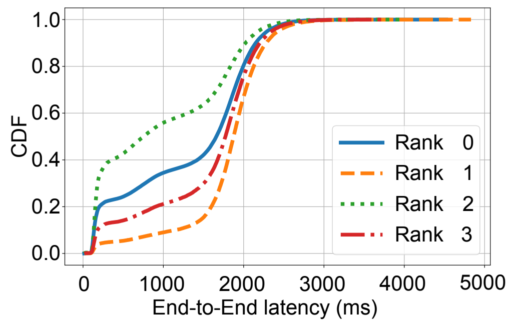
LatenSeer's design aims to solve latency estimation for real-world usages at scale. We discuss some design choices and optimizations that make LatenSeer practical, especially from our experiences when dealing with massive trace data at Twitter.

(1) Handling clock skew and miss data. To derive the serial and parallel relationships between nodes, we must compare the timestamps on two spans. These spans often represent RPCs that are likely emanating from different machines, which poses a potential issue as the clocks across these machines are not perfectly synchronized, thus could introduce inaccuracies in our comparisons. We mitigate this issue by using client-side timestamps, which record the times on the same machine where the RPC calls are invoked. In other words, the start and finish times of child spans that we use are the timestamps recorded from a machine where the parent span locates. In this manner, the duration of a child span consists of both its processing time and network latency. Handling missing or erroneous data in tracing systems is an open problem. We tackle this by truncating traces at the initial missing span connection. Comparisons on Alibaba traces revealed minimal latency differences between complete and truncated traces.

(2) Set nodes for clustering traces. As mentioned in §3.2.2, we introduce the concept of *set node* to cluster together traces that exhibit similar invocation. We now



(a) Frequency of identical dependency graphs and set nodes.



(b) Latency distribution of different set nodes from same parent node.

Figure 3.7: Set node justification via Twitter traces.

provide a more detailed explanation behind this design decision. Given the diverse call patterns in large-scale systems, clustering traces based on identical dependency graphs is often impractical. For instance, a service invoking four `cache-get` operations and a service executing five `cache-get` operations would fall into separate clusters when using an identical clustering approach. Fig. 3.7a illustrates the frequency of identical dependency graphs and set nodes of a root node (a front-end service) in Twitter, derived from examining 190,087 traces with the same front-end API. The rank in Fig. 3.7a refers to the ranking of identical dependency graphs or set nodes based on their frequency. In this context, we focus exclusively on the first depth of the full graph: the root node and its immediate child nodes. Recall that the identical dependency graphs are those where child nodes exhibit precisely the same sequence of invocations, and the set node groups the traces which have the same set of child nodes. Fig. 3.7a reveals that the number of different identical dependency graphs is twice that of set nodes. This set node concept, therefore, offers a more practical and effective way of grouping traces for latency modeling in complex systems.

The adoption of set nodes is substantiated by their ability to identify similar call patterns in traces sharing the same set of child RPC calls. Conversely, different set nodes usually correspond to varying latency distributions, a consequence of the unique call paths each trace navigates, as exemplified in Fig. 3.7b. Further supporting this approach, we've observed that the top two ranks of identical service dependency graphs exhibit the same latency distributions and belong to the same set node (the finding is not shown in the paper due to space constraints.). This consistency lends further credibility to the concept of the set node, underscoring its practicality and relevance in understanding and modeling system latencies.

(3) Pre-merging *parallel spans*. The traces in large-scale applications tend to be complex, with a typical request touching tens to thousands of individual microservices, producing service dependency trees that are up to 20 levels deep. To reduce the complexity of computation, we presume some sibling spans are parallel and coalesce these spans into the same (regular) node. According to observations of a large number of Twitter traces, we found most simultaneous RPCs to cache and storage are parallel. For example, hundreds of `cache-get` happening simultaneously often have a parallel relationship. Pre-merging such spans significantly reduce the L-TREE complexity.

(4) Building LatenSeer trees in parallel. LatenSeer must be efficient enough to process an enormous volume of traces for large-scale applications. To facilitate this, we parallelize the tree-building process. The traces are initially partitioned based on their top-level API endpoint, as identical API calls tend to exhibit similar calling patterns. Subsequently, traces within the same API endpoint are evenly distributed into the same bucket, and a subtree is constructed for each batch of traces. Finally, subtrees for traces with the same top-level API endpoint are merged. This merging process commences from the top and progresses downwards. When nodes are identical, they are merged along with their latency profiles. If nodes are distinct, the unique node is simply incorporated. This methodology ensures that the final tree accurately represents the diverse and complex calling patterns inherent in the large volume of trace data. We implemented L-Tree using a distributed data-parallel processing framework in Twitter. The framework can produce the results daily or weekly based on requirements.

3.3 Evaluation

In this section, we evaluate LatenSeer to answer the following questions.

- Can LatenSeer provide accurate end-to-end latency estimation results?
- How well is LatenSeer estimating the end-to-end latency with real-world use cases?
- How is LatenSeer compared with the state-of-the-art?
- How effectively can LatenSeer handle large-scale traces?

3.3.1 Experimental Setup

Prototype system. We implement LatenSeer[5] in $\approx$ 3,000 lines of Python3 code and test it with DeathStarBench microservice benchmark (DSB) [86] on *Social Network* application, which consists of 31 unique microservices. We leverage DSB's default workload generator to produce client requests. Furthermore, we fulfill all the missing function-level instrumentation in DSB.

We set up experiments on CloudLab [67] under two deployment topologies using three different physical sites, as shown in Fig. 3.8. We deploy the benchmark on 6 machines at the Utah site, 6 machines at the Wisconsin site, and 1 machine at the Clemson site. In both scenarios, we run the workload generator at the Clemson site on a node type "c6320" (Haswell 28-core with 256 GB RAM and 10 Gbps network). Microservices in Utah and Wisconsin ran on node types "xl170" (Broadwell 10-core with 64 GB RAM and 25 Gbps network) and "c220g5" (Skylake 20-core with 192 GB RAM and 10 Gbps network), respectively. RPC latencies within a single site are < 1 ms, while latencies between Utah and Wisconsin are in the range 38–42 ms.

Methods. Both the traces and our measurements report the end-to-end request processing latency at the API gateway. We call the latency distribution that LatenSeer

[5]LatenSeer is open-sourced at `https://github.com/yazhuo/LatenSeer`, and the Twitter traces will be released upon legal approval.

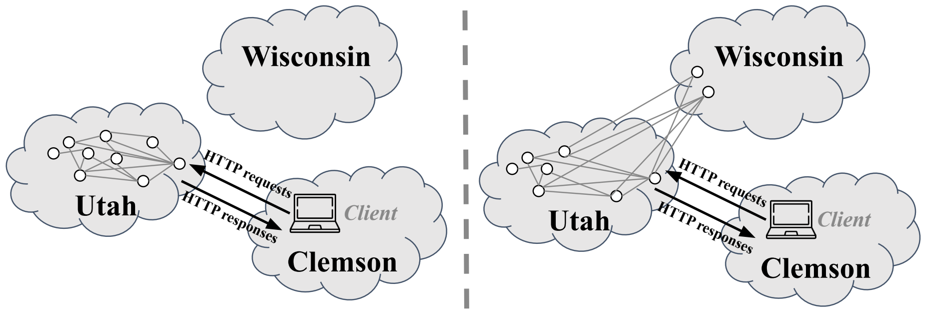


Figure 3.8: Prototype experiment setup using three Cloudlab sites. The left shows the single-site deployment where all microservices run in the Utah cluster; the right shows multi-site deployment where some microservices run in the Wisconsin cluster.

outputs for a specific service change scenario the *prediction*. We term a measured latency distribution for a given scenario *ground truth* and use it to quantify the accuracy of the corresponding prediction. When we exercise LatenSeer to make a prediction without perturbing the model latencies (typically to validate the model itself), we call the output latency distribution the *null prediction*.

Except for the sensitivity experiment that varies the request mix, the workload generator sends requests to three endpoints using the default read-dominated ratio of 1:3:6, with one `compose-post` request for every three calls to `read-user-timeline` and six `read-home-timeline` requests. We use the same request rate of 500 RPS in all cases.

Each experiment proceeds as follows. We first run our workload for 10 minutes in the baseline configuration to collect the traces for building the model in LatenSeer. Next, we inject an estimated latency delta (specifically, the average value from a ping test) into the model at the appropriate microservices and invoked `scenario.predict` to generate a latency prediction. We then run the workload again on the changed deployment for 10 minutes in order to measure the ground truth latency distribution. We compare prediction results with the ground truth. All traces are collected using Jaeger at a 10% sampling rate.

Metrics. For a prediction latency CDF F_{pred}, and a ground truth latency CDF F_{true}, we report prediction accuracy using two metrics:

1. The D-statistic in the K-S (Kolmogorov-Smirnov) test, which is the difference between two CDF curves at the point of maximum divergence. It is defined as $D = \max_x |F_{\text{pred}}(x) - F_{\text{true}}(x)| \in [0, 1]$.

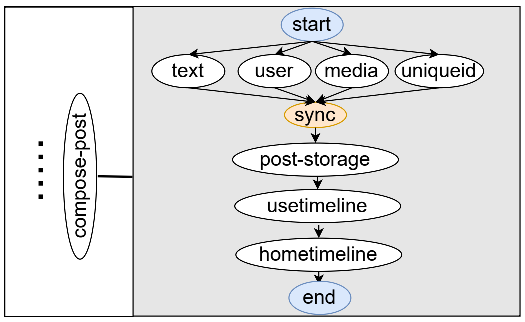
2. The maximum and average of the relative latency error at each percentile. It is defined as $E = \big((F^{-1}_{\text{pred}}(y) - F^{-1}_{\text{true}}(y)\big)$ $/F^{-1}_{\text{true}}(y) \in \mathbb{R}$, where $y \in [0, 1]$. A negative value implies that the prediction is lower than the ground truth; a positive value means that the prediction is higher latency than the ground truth. We focus on the mean and max relative error, denoted as E_{avg} and E_{max}.

The K-S statistic is a standard metric for comparing two CDFs, but because it captures only the most extreme point of misprediction, it can be large even when the two CDFs are mostly similar. In practice, it is also useful to know how well LatenSeer predicted latency across the entire distribution relative to the absolute values of the ground truth. Therefore we also report the relative error metric.

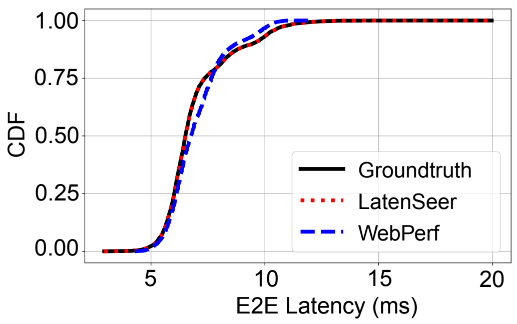
Traces. To evaluate scalability of LatenSeer in handling production-scale traces, we use two production traces from Twitter and Alibaba. The Twitter traces consist of up to 25,000 spans, with a depth spanning from 2 to 18 hops. The Alibaba traces possess up to 6,625 spans with 1 to 14 hops.

For latency estimation experiments, we operate our prototype system and gather traces using Jaeger [102] as part of the DSB deployment. The collected DSB trace set encompasses a variable number of spans, fluctuating between 5 and 33, with the maximum depth ranging from 2 to 6 RPC hops.

Baseline. We compare LatenSeer with WebPerf [105], the state-of-the-art latency estimation work. We reimplement WebPerf's model because it's not open-sourced, and tame WebPerf's model to fit our scenarios. WebPerf is based on customized low-level



(a) Ground truth



(b) LatenSeer vs. WebPerf

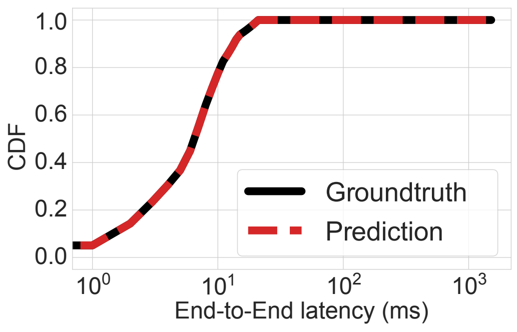
Figure 3.9: LatenSeer compared with state-of-the-art.

instrumentation for the extraction of causal dependency graphs, which we find not practical in today's tracing framework usages in most production environments. We resort to examining the DSB codebase to discern these dependencies and construct the corresponding graph to meet WebPerf's requirements.

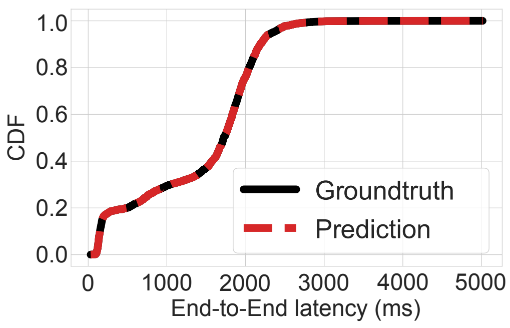
3.3.2 Estimation Accuracy

Through code reading for DSB benchmark, we derive an true causal dependency graph. This graph is then compared with the L-TREE generated by LatenSeer. Our analysis shows that LatenSeer accurately replicated the same dependency relationships present in the true graph. Fig. 3.9a presents a section of the L-tree at its highest complexity, which also aligns with the trace depicted in Fig. 2.1.

We then conduct a *null prediction* experiment to validate the soundness of our model. In this setup, we inject a "null" latency into the leaf nodes, triggering the latency propagation procedure to traverse all the nodes in the tree. We expect that the prediction aligns closely with the ground truth. As shown in Fig. 3.9b, LatenSeer precisely calculates the end-to-end latency distribution, while WebPerf generates larger errors with $D = 15\%$, $E_{max} = 8.8\%$ and $E_{avg} = -2.0\%$. The superior accuracy of LatenSeer over WebPerf can be attributed to the assumptions made by each tool regarding latency. WebPerf operates under the assumption that latencies on different



(a) Alibaba



(b) Twitter

Figure 3.10: Null prediction on production traces.

components are independent of one another, while LatenSeer takes a more holistic approach by profiling the latencies jointly for sibling nodes.

Since the ground truth of causal dependency graphs for the production traces is not available, we limit our examination to whether the model accurately predicts the latency distribution of the input traces through null prediction experiments on two production traces. The results in Fig. 3.10 show that predictions align closely with the ground truth for both Alibaba traces (Fig. 3.10a) and Twitter traces (Fig. 3.10b), confirming that the internal service relationships are modeled faithfully.

3.3.3 Case Studies

We evaluate how well LatenSeer models end-to-end request latency in a microservices environment using DSB. To examine the key properties, we focus on the two use cases: service placement (UC1) and latency slack (UC2).

Service placement (UC1)

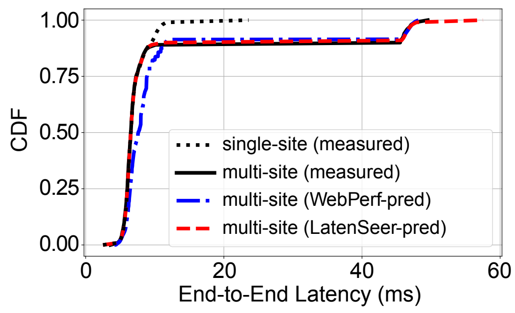
We study the accuracy of latency predictions under real, albeit controlled, conditions by comparing the predicted and measured latency distribution following wide-area service migration. In order to experimentally exercise as much of LatenSeer's model as possible, we use three API endpoints: `read-home-timeline`, `read-user-timeline`, and `compose-post`, and select three microservices for migration: `user-timeline-service`, `user-service`, and `media-service`.

- The `read-home-timeline` endpoint does not interact with any of the chosen services, hence unaffected by their migration.
- The `read-user-timeline` endpoint only interacts with `user-timeline-service`.
- The `compose-post` endpoint interacts with all three services, and invokes them both in parallel (`user-service` and `media-service`) and serially (`user-service` and `user-timeline-service`).

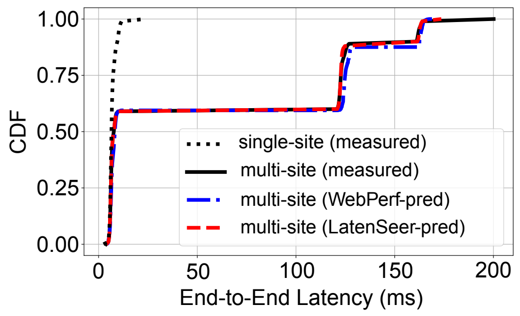
In the following experiments, we explore how accurately LatenSeer predicts end-to-end latency when selected microservices are migrated over the wide area network from one CloudLab site to another. We set up the experiments as described in §3.3.1, generating the model from the single-site deployment, measuring ground truth in the multi-site deployment, and using the average ping time measurement of 38.7 ms between the Wisconsin and Utah sites as the injected latency delta.

Migration that increases latency. Fig. 3.11 shows the prediction vs ground truth CDFs for experiments with two different pairs of services, each graph showing the top-level request latency distribution measured at the single-site (dotted line), the ground truth measured with the multi-site deployment (solid line), the prediction from WebPerf (dash-dot line), and the prediction from LatenSeer (dashed line). For the left-hand graph, Fig. 3.11a, the `user-service` and `media-service` were moved from Utah to the Wisconsin site. Only one endpoint, `compose-post`, should be affected by this migration. This endpoint comprises 10% of the default workload mix, and the two microservices have a parallel relationship within requests for that endpoint.

Fig. 3.11b shows the prediction when `user-timeline-service` and `user-service` were migrated. These microservices affect not only `compose-post`, with a serial rela-



(a) Migration of parallel services, affecting a single endpoint



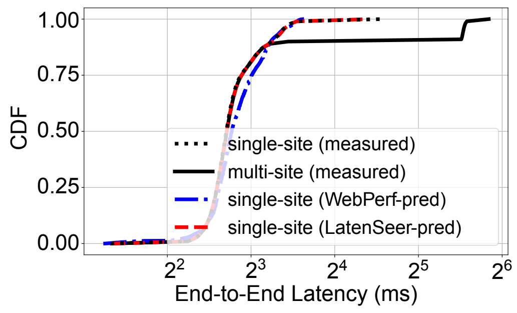
(b) Migration of serial services, affecting two endpoints

Figure 3.11: Prediction accuracy for migrating from local to remote.

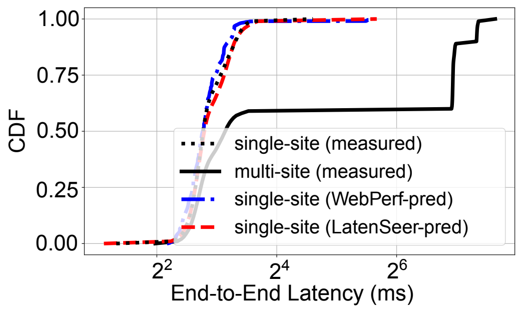
tionship therein, but also impact `read-user-timeline`, which is called 30% of the time. Note that in both cases, latency is extended by different amounts depending on the number of cross-site calls introduced by the service migration.

LatenSeer shows highly accurate predictions for both scenarios, as summarized in Table 3.1. Even though the K-S D statistic for LatenSeer for the right-hand graph is almost 5%, the average relative error is extremely low at -0.11%. In comparison, WebPerf shows much lower accuracy with 19% and -1.22% for D statistic and average relative error, respectively. Note how the shape of the CDF changes when some services are placed remotely: the jumps reflect the workload mix and how the three request types are affected (or not) by the migration. Thus Fig. 3.11a has just one jump at around the 90th percentile (because 10% of requests are `compose-post`), while Fig. 3.11b shows two jumps for `read-user-timeline` and `compose-post`, both of which touch the migrated services. The change in the shape of the latency distribution highlights that one cannot simply estimate the latency impact of service migration by offsetting the baseline distribution with a constant value and emphasizes the importance of LatenSeer's modeling techniques.

Migration that decreases latency. This experiment inverts the previous: we build the model from traces collected using the multi-site deployment and predict the single-site end-to-end latency distribution by injecting the negative delay value



(a) Parallel services migration, affecting a single endpoint



(b) Serial services migration, affecting two endpoints

Figure 3.12: Prediction accuracy for migrating from remote to local.

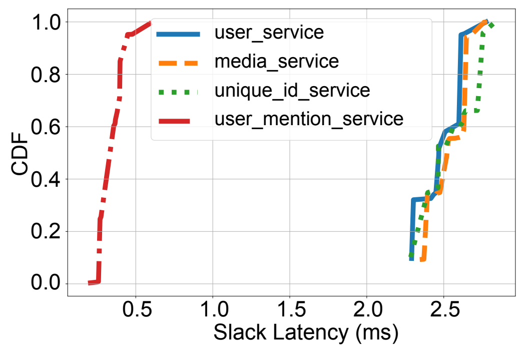
Table 3.1: Results of service placement experiments.

Experiments	**Model**	$D(\%)$	$E_{\mathbf{max}}(\%)$	$E_{\mathbf{avg}}(\%)$
Parallel services	LatenSeer	1.31	2.30	-0.21
→ remote (Fig. 3.11a)	WebPerf	18.60	27.00	9.30
Serial services	LatenSeer	4.68	11.50	-0.11
→ remote (Fig. 3.11b)	WebPerf	19.04	28.00	-1.20
Parallel services	LatenSeer	1.45	1.10	0.22
→ local (Fig. 3.12a)	WebPerf	10.50	10.70	0.60
Serial services	LatenSeer	5.35	8.00	1.50
→ local (Fig. 3.12b)	WebPerf	9.50	8.90	-0.80

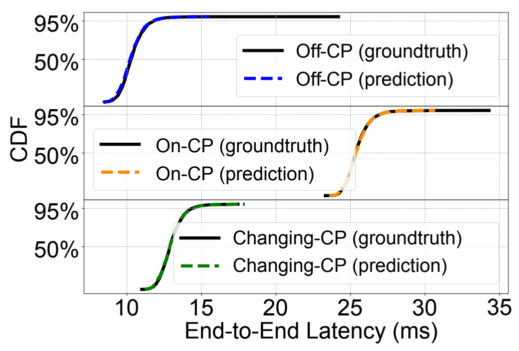
(-38.7ms) to the target nodes. Fig. 3.12 shows the results; the left-side plot delineates the effects on moving the parallel services (`user-service` and `media-service`), while the right-side plot shows the results of moving serial services (`user-service` and `user-timeline-service`). Once again, the prediction accuracy is very good, with average relative errors for both experiments below 2% and D-statistic less than 5.35%. As a comparison, WebPerf shows errors of more than 9.5%.

Slack Analysis (UC2)

LatenSeer can be used to infer the latency budget of specific microservices: it traverses the L-TREE top-down, from the root node to all leaves, to compute the available latency slack for services. Fig. 3.13a shows four services with latency slack under our experimental workload: `unique-id-service`, `user-service`, and `media-service`



(a) Latency slack distributions output from LatenSeer



(b) Latency prediction from perturbation guided by latency slack

Figure 3.13: Latency slack analysis.

have similar latency slack distributions — three of the services exhibited at least 2.3 ms slack, while the latency slack for `user-mention-service` is much smaller. Other services, not shown, had zero latency slack, meaning they were on the latency-critical path, and any increase in latency of these services would affect end-to-end latency.

We evaluate the accuracy of the latency slack identified by LatenSeer at individual services, using the slack to systematically perturb the latency distributions of services on and off the latency-critical path. For these experiments, we use the single-site deployment configuration and perturb latency in a controlled fashion by using the `tc` utility to induce delay at the network level of the Docker container of the target service. The same delay value is injected into the target service in LatenSeer's model, which then updates the latency slack at each node in a top-down fashion.

We evaluate the accuracy of slack estimates in each of the three ways that injected latency can perturb the L-TREE: (i) when the injected latency is off the latency-critical path; (ii) when the injected latency extends or reduces the duration of the latency-critical path; and (iii) when the injected latency changes the latency-critical path itself.

OFF-CP: For the first prediction experiment, we inject 2 ms of latency – well within the slack time – at `user-service` that is off the latency-critical path. Fig. 3.13b "Off CP" indeed confirms this, with the prediction almost unchanged from ground

Table 3.2: Results of sensitivity analysis on the injected latency for the experiment shown in Fig. 3.11a.

Diff from ping time(%)	D(%)	E_{max}(%)	E_{avg}(%)
-10	9.5	2.3	-0.96
-5	7.8	2.3	-0.58
-2	3.5	2.3	-0.36
-1	1.8	2.3	-0.29
+1	2.5	2.3	-0.01
+2	4.6	2.3	-0.06
+5	8.0	4.4	0.16
+10	9.6	8.6	5.40

truth ($D = 4.26\%, E_{max} = -1.0\%, E_{avg} = -0.65\%$).

On-CP: We then inject 5 ms of latency to `user-timeline` service that is inferred to be on the latency-critical path (zero latency slack). The new dominating latency distribution bubbles up through the levels of the L-tree to the root node. Fig. 3.13b "On CP" shows the results with $D = 2.01\%, E_{max} = 0.7\%, E_{avg} = -0.11\%$.

Changing-CP: For the third injection experiment, we change the critical path by again adding latency to `user-service`. In this case, however, the injected latency of 5 ms exceeds the available slack, causing the change in end-to-end latency. We show the results in Fig. 3.13b "Changing CP". Note that this experiment injects the same magnitude of latency change as the previous one (albeit into different microservices), but the resulting distribution is markedly different, and moreover, LatenSeer successfully predicts this difference ($D = 0.02\%, E_{max} = 0.3\%, E_{avg} = 0.1\%$).

3.3.4 Sensitivity Analysis

We conduct a sensitivity analysis on injected latency and on changes to the workload request mix, showing how the metrics degrade as the model diverges further from conditions experienced in the prediction environment.

Injected latency. In the service migration experiments, we use the average ping

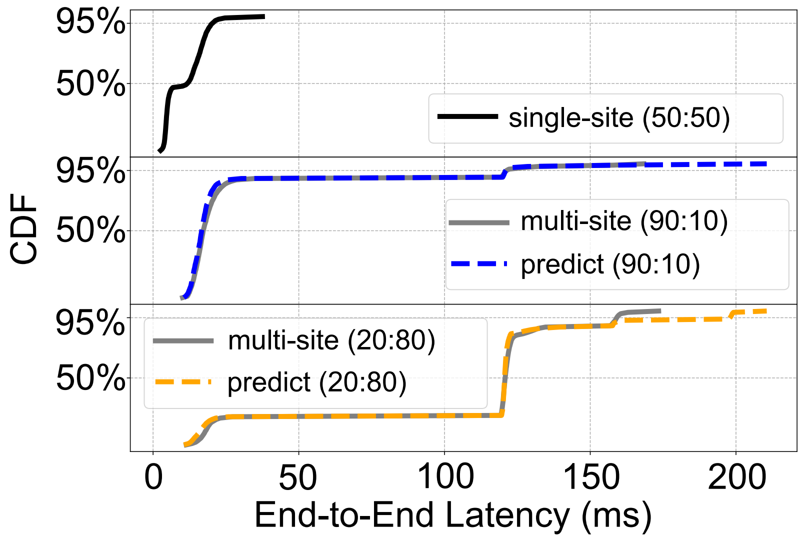


Figure 3.14: Latency prediction. Sensitivity to different workload distributions.

time between two sites to approximate the additional latency introduced by the new placement. This is obviously a low-fidelity value – single packet timings at the network layer are not the same as timings of RPC over TCP, across a WAN link, with varying payload size. However, we also claim that this is a realistic starting point in an industry setting, offering a simple-to-obtain, ”good enough” value for many real-world scenarios that require only an approximate answer to a what-if question.

To better characterize the impact of such inaccuracy, we repeat the first service placement experiment with various latency injections as fractions of the ping value. Table 3.2 shows the results: the K-S statistic, D, in particular, reports increasingly large divergences between prediction and ground truth, although overall relative error does not vary much from the baseline we use in the experiments reported above.

Workload mix. LatenSeer predicts latency using the *historical* data. In the real world, it is not unusual for the mix of request types to change over time, and so we look here at how varying the relative request proportions affects the quality of the prediction. In this experiment, we first collect traces in the *single-site* scenario, which comprises a 50:50 mixture of request types `read-home-timeline` and `read-user-timeline`. Then, we migrate `user-timeline-service` to the multi-site scenario and collect

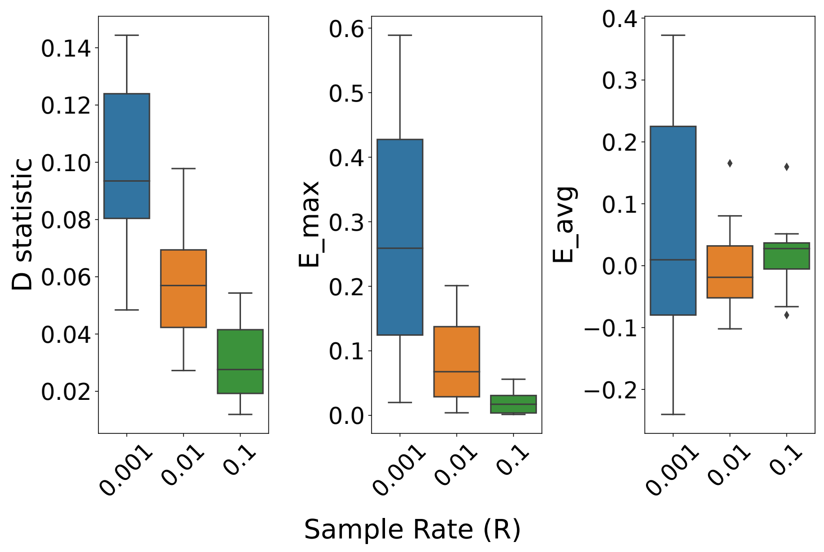


Figure 3.15: Error analysis. Relative errors of prediction for different sampling rates.

ground truth using 90:10 and 20:80 mixtures.

Fig. 3.14 shows the model's accurate predictions up to approximately the 90th percentile, but results degrade at the tail. This decline is attributed to a higher cache hit ratio during skewed workloads, a detail not covered by LatenSeer's modeling, hence the anticipated inaccuracy.

Threshold for succession time distribution. As discussed in §3.2.2, we choose the median value from the succession time CDF as our classification for all experiments. Here we repeat the first service placement experiment, adjusting the classification threshold between 1% and 99% to extract causal dependencies. Our results indicate same values for D, E_{max}, and E_{avg} across all classification thresholds. This consistency can be attributed to the relatively low level of noise present within the benchmark environment. However, it's important to recognize that this threshold may not be universally applicable. It is recommended that the threshold be evaluated and potentially adjusted to suit the unique conditions of different production environments.

Varying trace sampling rate. As tracing systems only capture a subset of requests, we evaluate the accuracy of prediction with different sampling rates over 4 sets of migration experiments. We randomly sample the traces that are used to build the model in the previous experiments as *sample* traces, then we use the *sample* traces to build the model and conduct prediction. Unless otherwise stated, we repeat this experiment 20 times for each sampling rate.

The box plots in Fig. 3.15 show the error metrics (D, E_{max}, and E_{avg}) between prediction from *sample* traces and ground truth for different sample rates. Overall, the D statistic is small, even for low sampling rates. Sampling with R = 0.001 results in E2E latency prediction with D statistic of between 0.05 and 0.14. E_{max} and E_{avg} show more over-prediction when the sampling rate is too low.

3.3.5 LatenSeer Performance

Building the LatenSeer model, or L-TREE, is dependent on the internal structure of the aggregated trace tree. In controlled experiments, we have found that constructing the LatenSeer from 30,000 DSB traces took approximately 11 minutes. The number of spans in our DSB trace set ranged from 5-33, and the depth extended from 2-6 RPC hops. However, this timing can significantly vary in real-world settings, owing to the complexity of the traces involved.

For instance, the number of spans in our DSB trace set ranged from 5-33, and the depth extended from 2-6 RPC hops. This complexity is dwarfed when we consider Alibaba and Twitter traces. Alibaba traces can contain up to 6,625 spans with depths between 1-14 hops. In contrast, Twitter traces can encompass up to 25,000 spans and depths ranging from 2-18 hops. Given these complexities, the time to build the L-TREE in real-world scenarios can be considerably longer. For 71,055 Alibaba traces, the model-building process requires approximately 66 minutes. Similarly, for 2,618 traces from Twitter, the construction process takes about an hour.

3.4 Discussion

End-to-end latency estimation is important to engineering planning, performance analysis and optimization, however, the traditional approaches are deficient. To explain, we examine the main strategies deployed in practice.

Canarying releases. The canary release strategy, widely utilized in production, facilitates risk reduction by incrementally introducing software updates to a limited user group before broader deployment [207]. With scarce engineering resources, operators wish to conduct latency estimation *before* expensive engineering efforts are spent on the project. Such a feasibility review should precede any live traffic analysis or A/B experimentation—canarying—on the resulting component [199] to minimize cost and customer impact of potentially poor design decisions.

Service dependency graphs. A common technique for understanding the causal dependencies between services is to chart service dependencies, a method widely adopted in production [108, 128, 129, 178, 200]. Large-scale applications requests are represented as call paths through *service dependency graphs*, which capture communication-based dependencies between microservice instances. The call paths show how requests flow among microservices by following parent-child relationship chains. As such, service dependency graphs serve an important tools in discerning the complex interplay of services and optimizing application performance.

Many distributed tracing visualization tools [102, 108, 178, 244] aggregate traces to construct service dependency graphs at various detail levels. Yet, none, to our knowledge, have integrated latency distribution into the service dependency graphs. Introducing latency estimation to a microservice-based dependency graph faces hurdles. Alibaba [128] shows that the latency of a service is stable among call graphs of similar topologies but varies significantly across different topologies, and the latency of a service is stable when the call rate varies. Some works [129, 200] use dependency graphs to guide auto-scaling and service migration. Tprof [98] aggregated traces in a fine-grained manner of sub-span analysis for performance debugging.

While a service dependency graph outlines relationships between services, it fails to detail the micro-level dynamics of execution order, including the nature of sibling relationships, whether parallel or sequential. For instance, the service dependency graph in Fig. 2.1 illustrates service `compose-post` must await the responses from all seven child services to complete its task. Yet, if we aim to comprehend how changes in the `media` service affect `compose-post`, the service dependency graph falls short. This limitation impedes accurate latency estimation, posing a risk of relying on models grounded in potentially incorrect assumptions about the interrelationships of child services.

Latency-critical paths. In microservice dependency graphs, the latency-critical path represents the slowest sequence of dependent tasks [217]. While its analysis facilitates the identification of optimization prospects and bottlenecks in distributed systems [31, 43, 53, 68, 80, 108, 161, 235], it can inadvertently obscure potential issues in off-path services. Hence, incorporating *slack* time analysis for off-path services is advocated, aiding in refined capacity planning and mitigating risks arising from shared-resource contention [68].

CRISP [235] uses critical path analysis on Uber traces to help developers understand and optimize important services. However, services off the latency-critical path are easily trimmed from the results. For example, service `media` in Fig. 2.1 are not shown in the final results of CRISP. A singular focus on the latency-critical paths paints an incomplete and misleading picture for end-to-end latency estimation, emphasizing the need for a more holistic approach that incorporates the dynamic roles of all services in the system.

3.5 Related Work

Gleaning causal dependencies in existing large-scale systems, such as building causal models of latency, has been explored in the past decade. Researchers from Google, for example, built theoretical frameworks to understand the latency profile of arbitrary black box services [114, 153], which, like more recent work over microservices [126], is focused around anomaly detection. More recent white-box approaches to performance modeling have also been proposed [20, 163].

Facebook's Mystery machine [53] shares the motivation of constructing a causal model using pre-existing data, predating modern tracing infrastructure, that works by initially hypothesizing all possible pairwise relationships and gradually rejecting the causality of each dependency through counterexamples. Regrettably, the ensuing predictions are brittle owing to incomplete methods for producing causal diagrams [107] and cannot handle various issues in tracing data, such as clock skew and missing span. CRISP [235] shares a similar motivation **(G1)** but lacks a causal treatise of latency estimation. Similarly, Orion [134] also uses latency propagation technique to calculate end-to-end latency. However, it focuses on known service DAGs and fails to address the causality in service dependencies. The study most closely aligned with our objectives is WebPerf [105], which crafts techniques surrounding service dependencies and latency propagation. However, WebPerf is specifically tailored to Microsoft's Azure environment and relies on Azure-specific hints to work. WebPerf employs domain-specific binary instrumentation, presenting a solution that isn't universally adaptable. In contrast, LatenSeer advances this approach by adopting a data-driven strategy, relying on trace data that are commonly available in today's production systems.

Latency prediction is a well-trodden field with substantial research in areas such as service selection [76, 229], service composition [21, 22, 24, 49, 233], and business process modeling [61, 171]. Notably, with the rising popularity of serverless workflows in many

Table 3.3: LatenSeer compared to state-of-the-art systems.

	LatenSeer	ORION	CRISP	WebPerf	MysteryMachine
G1	✓		✓		✓
G2	✓	✓		✓	✓
G3	✓	✓		✓	

applications, there has been a surge in research efforts aimed at latency prediction for these workflows. These efforts predominantly focus on resource optimization and reducing communication latency in serverless workflows [74, 75, 135]. Distinctively, LatenSeer, not constrained by predefined workflows, offers a flexible solution to latency predictions in microservice-based applications.

3.6 Conclusion

We present LatenSeer, a data-driven modeling framework for estimating end-to-end latency distributions in microservice-based web applications. LatenSeer can accurately predict interventional end-to-end latency by leveraging distributed tracing data. We evaluated LatenSeer in two realistic scenarios: service placement and latency slack analysis. Our evaluation shows that LatenSeer achieves high precision accuracy with an estimation error less than 5.35% (D-statistic), outperforming the start-of-the-art that has more than 9.5% estimation error. Moreover, our results on real-world production traces show that LatenSeer is practical and scalable enough to support complexities in production environments.

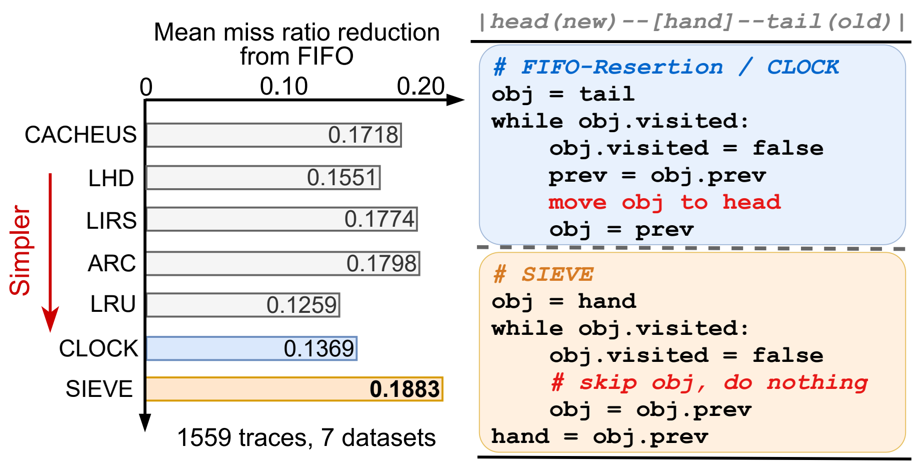
Chapter 4

SIEVE: an Efficient Turn-Key Eviction Algorithm for Web Caches

4.1 Introduction

Web caches, such as Content Delivery Networks (CDNs) and key-values caches, are widely deployed in today's digital landscape to reduce user request latency [28, 39, 40, 56, 147, 164, 179, 228], network bandwidth [97, 99, 182, 223], repeated computation [47, 212, 225, 226]. As a critical component of modern infrastructure, these caches often have a large footprint. For example, Netflix used 18,000 servers for caching over 14 PB of application data in 2021 [140]; while Twitter reportedly had 100s of clusters using 100s TB of DRAM and 100,000s of CPU cores for in-memory caching in 2020 [224].

At the heart of a cache is the eviction algorithm, which plays a crucial role in managing limited cache space. Such algorithms are efficient when they can retain more valuable objects in the cache to achieve a lower miss ratio—the fraction of requested objects that must be fetched from the backend. The quest for high efficiency has spurred a long repertoire of clever algorithms, but most, if not all, trade off simplicity in exchange for efficiency gains. For example, ARC [139], SLRU [99], 2Q [106],



```
|head(new)--[hand]--tail(old)|
# FIFO-Resertion / CLOCK
obj = tail
while obj.visited:
    obj.visited = false
    prev = obj.prev
    move obj to head
    obj = prev
# SIEVE
obj = hand
while obj.visited:
    obj.visited = false
    # skip obj, do nothing
    obj = obj.prev
hand = obj.prev
```

Figure 4.1: SIEVE is simple and efficient. The code snippet shows how FIFO-Reinsertion and SIEVE find eviction candidates. Minor code changes convert FIFO-Reinsertion to SIEVE, unleashing lower miss ratios than state-of-the-art algorithms.

and MQ [243] manage multiple least-recently-used (LRU) queues to achieve better efficiency. LHD [33], CACHEUS [170], LRB [182], and GL-Cache [219] use machine learning techniques that further increase system and lookup complexity. Furthermore, many of these algorithms require explicit or implicit parameter tuning to achieve good efficiency on a target workload.

The conventional wisdom among systems operators is that *simple is beautiful*: simplicity is a key appealing feature for an algorithm to be deployed in production since it commonly correlates with effectiveness, maintainability, scalability, and low overhead. To illustrate, note that most caching systems or libraries in use today, such as ATS [23], Varnish [198], Nginx [146], Redis [8], groupcache [44], use only FIFO and LRU policies.

Inspired by recent observations that cache items slated for one-time use often remain too long in LRU or FIFO-style caches [221], we discovered SIEVE, a simple algorithm that achieves high efficiency across a wide range of web cache workloads. Fig. 4.1 shows SIEVE's simplicty compared to a decades-old algorithm, FIFO-Reinsertion. Instead of moving the to-be-evicted object that has been accessed to the head of queue, we retain it in its original location. We implemented SIEVE in five production cache libraries, which required fewer than 20 lines of change on average, underscoring the ease of real-world deployment.

Despite a simple design, SIEVE can quickly remove unpopular objects from the cache, achieving comparatively high efficiency compared to the state-of-the-art algorithms. By experimentally evaluating SIEVE on 1559 traces from five public and two proprietary datasets, we show that SIEVE achieves similar or higher efficiency than 9 state-of-the-art algorithms across traces. Compared to ARC [139], SIEVE reduces miss ratio by up to 63.2% with a mean of 1.5% [1]. As a comparison, ARC reduces LRU's miss ratio by up to 33.7% with a mean of 6.7%. Moreover, compared to the best of all algorithms, SIEVE has lower miss ratio on over 45% of the 1559 traces. In comparison, the runner-up algorithm only outperforms other algorithms on 15% of the traces.

SIEVE's design eliminates the need for locking during cache hits, resulting in a boost in multi-threaded throughput. Our prototype implementation in Cachelib [63] demonstrates that SIEVE achieves twice the throughput of an optimized LRU implementation when operating with 16 threads.

Through empirical evidence and analysis, we illustrate that SIEVE's efficiency stems from sifting out unpopular objects over time. SIEVE transcends a single standalone algorithm — it can also be embedded within other cache policies to design more advanced algorithms. We demonstrate the idea by replacing the LRU components in ARC, TwoQ, and LeCaR with SIEVE. The SIEVE-supported algorithms significantly outperform the original LRU-based algorithms. For example, ARC-SIEVE reduces ARC's miss ratio by up to 62.5% with a mean of 3.7% across the 1559 traces.

Our work makes the following contributions.

- We present the design for SIEVE: an easy, fast, and surprisingly efficient cache eviction algorithm for web caches.

- We demonstrate SIEVE's simplicity by implementing it in five production cache

[1]Due to a large number of traces, the mean miss ratio looks small.

libraries by changing less than 20 lines of code on average.

- Using 1559 traces from 7 datasets, we show that SIEVE outperforms all state-of-the-art eviction algorithms on more than 45% of the traces.
- We illustrate SIEVE's scalability using our Cachelib-based implementation, which achieves 17% and 125% higher throughput than optimized LRU at 1 and 16 threads.
- We show how SIEVE, as a turn-key cache primitive, opens new opportunities for designing advanced eviction algorithms, e.g., replacing the LRU in ARC, TwoQ, and LeCaR with SIEVE.

4.2 Design and Implementation

4.2.1 Sieve Design

In this section, we introduce SIEVE, a cache eviction algorithm that achieves both simplicity and efficiency.

Data structure. SIEVE requires only one FIFO queue and one pointer called "hand". The queue maintains the insertion order between objects. Each object in the queue uses one bit to track the visited/non-visited status. The hand points to the next eviction candidate in the cache and moves from the tail to the head. Note that, unlike existing algorithms, e.g., LRU, FIFO, and CLOCK, in which the eviction candidate is always the tail object, the eviction candidate in SIEVE is an object somewhere in the queue.

Sieve operations. A cache hit in SIEVE changes the visited bit of the accessed object to 1. For a popular object whose visited bit is already 1, SIEVE does not need to perform any operation. During a cache miss, SIEVE examines the object pointed by the hand. If it has been visited, the visited bit is reset, and the hand moves to the next position (the retained object stays in the original position of the queue). It continues this process until it encounters an object with the visited bit being 0, and it evicts the object. After the eviction, the hand points to the next position (the previous object in the queue). While an evicted object is in the middle of the queue most of the time, a new object is always inserted into the head of the queue. In other words, the new objects and the retained objects are not mixed together.

We detail the algorithm in Alg. 2, and we show a running example at `https://sievecache.com`. Line 1 checks whether there is a hit, and if so, then line 2 sets the visited bit to one. In the case of a cache miss (Line 3), Lines 5-12 identify the object to be evicted.

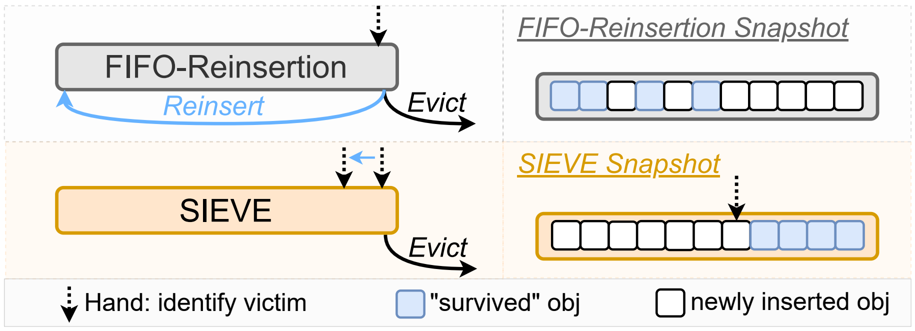


Figure 4.2: An illustration of SIEVE. Note that FIFO-Reinsertion and CLOCK are different implementations of the same algorithm. We use FIFO-Reinsertion in the illustration but will use CLOCK in the rest of the text because it is more commonly used and is shorter.

Algorithm 2 SIEVE

```
Require: The request $x$, doubly-linked queue $T$, cache size $C$, hand $p$
 1: if $x$ is in $T$ then
 2:     $x$.visited $\leftarrow 1$
 3: else
 4:     if $|T| = C$ then
 5:         $o \leftarrow$ p
 6:         if $o$ is NULL then
 7:             $o \leftarrow$ tail of $T$
 8:         while $o$.visited $= 1$ do
 9:             $o$.visited $\leftarrow 0$
10:             $o \leftarrow o$.prev
11:             if $o$ is NULL then
12:                 $o \leftarrow$ tail of $T$
13:         $p \leftarrow o$.prev
14:         Discard $o$ in $T$
15:     Insert $x$ in the head of $T$.
16:     $x$.visited $\leftarrow 0$
```

At first glance, SIEVE is similar to CLOCK/Second Chance/FIFO-Reinsertion [2]. Both maintain a single queue in which each object is associated with a visited bit to track its access status. Visited objects are retained (also called ”survived”) during an eviction. However, the hand in SIEVE moves from the tail to the head over time, whereas the hand in FIFO-Reinsertion stays at the tail. The key difference is where a retained object is kept. SIEVE keeps it in the old position, while FIFO-Reinsertion inserts it at the head, together with new objects. We illustrated this in Fig. 4.2.

[2]Note that Second Chance, CLOCK, and FIFO-Reinsertion are different implementations of the same eviction algorithm.

Lazy promotion and quick demotion. Despite a simple design, SIEVE effectively incorporates both lazy promotion and quick demotion. As described in §2.2.3, an object is only promoted at the eviction time in lazy promotion. SIEVE operates in a similar manner. However, rather than promoting the object to the head of the queue, SIEVE keeps the object at its original location. The "survived" objects are generally more popular than the evicted ones, thus, they are likely to be accessed again in the future. By gathering the "survived" objects, the hand in SIEVE can quickly move from the tail to the area near the head, where most objects are newly inserted. These newly inserted objects are quickly examined by the hand of SIEVE after they are admitted into the cache, thus achieving quick demotion. This eviction mechanism makes SIEVE achieve both lazy promotion and quick demotion without adding too much overhead.

The key ingredient of SIEVE is the moving hand, which functions like an adaptive filter that removes unpopular objects from the cache. This mechanism enables SIEVE to strike a balance between finding new popular objects and keeping old popular objects. We discuss more in §4.4.

4.2.2 Implementation

Simulation. We built a cache simulator for comparing different eviction algorithms, and we have cross-compared it with libCacheSim [218]. Besides SIEVE, our simulator implements ARC [139], LIRS [104], CACHEUS [170], LeCaR [202], TwoQ [106], LHD [33], Hyperbolic [42], FIFO-Reinsertion/CLOCK [59], B-LRU (Bloom Filter LRU), LRU, LFU, and FIFO. For all state-of-the-art algorithms, we used the configurations from the original papers.

Prototype. Because of SIEVE's simplicity, it can be implemented on top of a FIFO, LRU, or CLOCK cache in just a few lines by adding, initializing, and tracking the "hand" pointer. The object pointed to by the hand is either evicted or retained, depending on whether it has been accessed.

We implemented Sieve caching in five different open-source cache libraries: Cachelib [38], groupcache [44], mnemonist [6], lru-dict [3], and lru-rs [4]. These represent the most popular cache libraries of five different programming languages: C++, Golang, JavaScript, Python, and Rust. All five of these production cache libraries implement LRU as the eviction algorithm of choice. Aside from mnemonist, which uses arrays, they all use doubly-linked-list-based implementations of LRU. Adapting these LRU implementations to use Sieve was a low effort, as mentioned earlier.

The code and data used in this work are open-sourced at `https://github.com/yazhuo/NSDI24-SIEVE`. This includes the simulator and the prototypes.

4.3 Evaluation

In this section, we evaluate SIEVE to answer the following questions.

- Does SIEVE have higher efficiency than state-of-the-art cache eviction algorithms?
- Can SIEVE improve a cache's throughput and scalability?
- Is SIEVE simpler than other algorithms?

4.3.1 Experimental Setup

Workloads. Our experiments use open-source traces from Twitter [225], Meta [9], Wikimedia [208], TencentPhoto [239, 240], and two proprietary CDN datasets. We list the dataset information in Table 4.1. It consists of 1559 traces, 247,017 million requests to 14,852 million objects. Notably, our research is centered around web traces. We replayed the traces in the simulator and the prototypes as a closed system with instant on-demand fill.

Metrics. Miss ratio serves as a key performance indicator when evaluating the efficiency of a cache system. However, when analyzing different traces (even within the same dataset), the miss ratios can vary significantly, making direct comparisons and visualizations infeasible, as shown in Fig. 4.3. Therefore, we calculate the miss ratio reduction relative to a baseline method (FIFO in this work): $\frac{mr_{FIFO}-mr_{algo}}{mr_{FIFO}}$ where mr stands for miss ratio. If an algorithm's miss ratio is higher than FIFO, we use $\frac{mr_{FIFO}-mr_{algo}}{mr_{algo}}$. This metric has a range between -1 and 1.

We measure throughput in millions of operations per second (Mops) to quantify a cache's performance. To evaluate scalability, we vary the number of trace replay threads from 1 to 16 and measure the throughput.

Testbed. We conducted all evaluations on Cloudlab [66]. We implemented SIEVE and the state-of-the-art eviction algorithms in libCacheSim [218]. The simulations

Table 4.1: Datasets used in this work. CDN 1 and 2 are proprietary, and all others are publicly available.

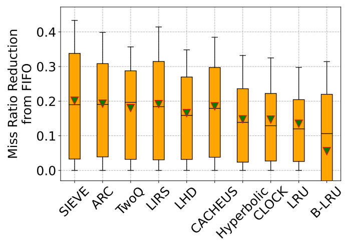
trace collections	approx time	# traces	cache type	# request (million)	# object (million)
CDN 1	2021	1273	object	37,460	2,652
CDN 2	2018	219	object	3,728	298
Tencent Photo [239]	2018	2	object	5,650	1,038
Wiki CDN [208]	2019	3	object	2,863	56
Twitter KV [225]	2020	54	KV	195,441	10,650
Meta KV [9]	2022	5	KV	1,644	82
Meta CDN [9]	2023	3	object	231	76

were executed on various types of nodes from the Clemson or Utah site, with the specific node types being dependent on their availability at the time of evaluation. Our prototype testbed used the c6420 node from the Clemson site, which has a dual-socket Intel Gold 6142 at 2.6 GHz with 384 GB DDR4 DRAM. We turned off turbo boost and pinned threads to CPU cores in one NUMA node in our evaluations. To demonstrate the accuracy our simulation, we compared the results of simulator and prototype on selected traces from various sources[3]. Despite observed differences in miss ratios, the general performance trends between the simulator and prototype were consistent. Notably, SIEVE consistently outperformed LRU in both setups, and its performance was similar to TinyLFU, especially with larger cache sizes. This consistency underscores the simulator's credibility in accurately mirroring the prototype's behavior.

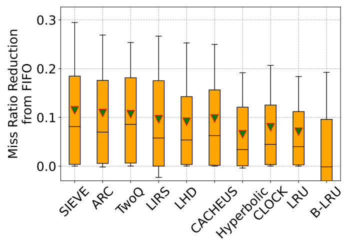
4.3.2 Efficiency Results

In this section, we compare the efficiency of different eviction algorithms. Because many caches today use slab-based space management, in which evictions happen on objects of similar sizes, we do not consider object size in this section. The cache sizes are determined as a percentage of the number of objects in a trace. We evaluate eight

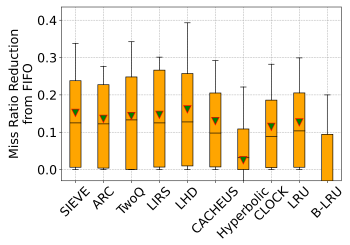
[3]Since running the prototype can be quite expensive, we opted to test only a random subset of the traces, rather than all of them.



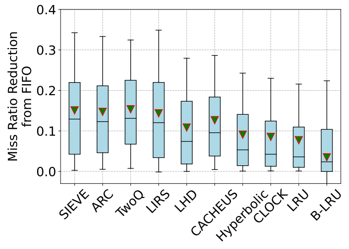
(a) CDN1 workloads, large cache, 1273 traces



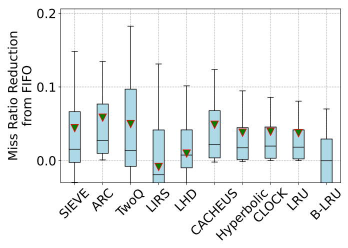
(b) CDN2 workloads, large cache, 219 traces



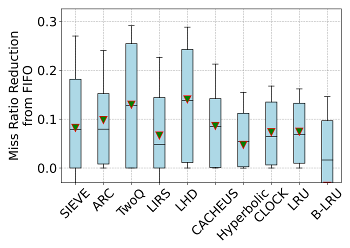
(c) Twitter workloads, large cache, 54 traces



(d) CDN1 workloads, small cache, 1273 traces



(e) CDN2 workloads, small cache, 219 traces

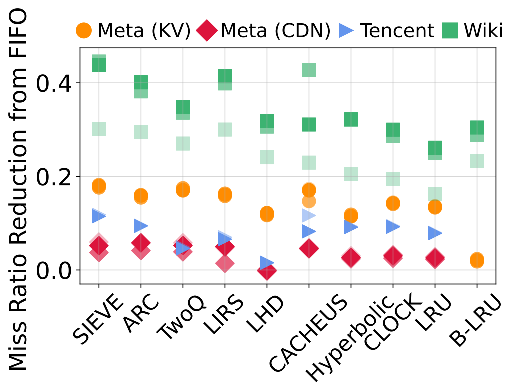


(f) Twitter workloads, small cache, 54 traces

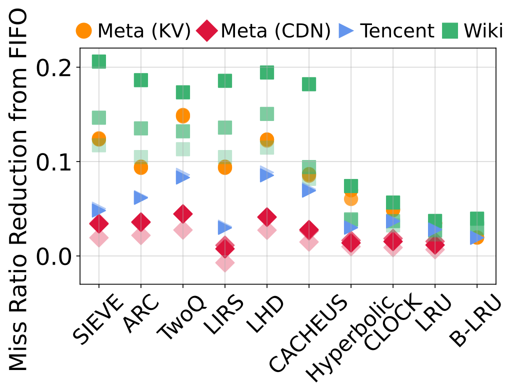
Figure 4.3: The box shows the miss ratio reduction from FIFO over all traces in the dataset. The box shows P25 and P75, the whiskers show P10 and P90, and the triangle shows the mean. The large cache uses 10% of the trace footprint, and the small cache uses 0.1% of the trace footprint. SIEVE achieves similar or better miss ratio reduction compared to state-of-the-art algorithms.

cache sizes using 1559 traces from the 7 datasets and present two representative sizes at 0.1% and 10% of the trace footprint (the number of unique objects in the trace).

Three large datasets CDN1, CDN2 and Twitter. Fig. 4.3 shows the miss ratio reduction (from FIFO) of different algorithms across traces. The whiskers on the boxplots are defined using p10 and p90, allowing us to disregard extreme data and concentrate on the typical cases. At the large cache size, SIEVE demonstrates the most significant reductions across nearly all percentiles. For example, SIEVE reduces FIFO's miss ratio by more than 42% on 10% of the traces (top whisker) with a mean of 21% on the CDN1 dataset using the large cache size (Fig. 4.3a). As a comparison, all other algorithms have smaller reductions on this dataset. For example, CLOCK/FIFO-Reinsertion, which is conceptually similar to SIEVE, can only reduce FIFO's miss ratio by 15% on average. Compared to advanced algorithms, e.g., ARC, SIEVE reduces ARC miss ratio by up to 63.2% with a mean of 1.5%. We remark that



(a) Large cache

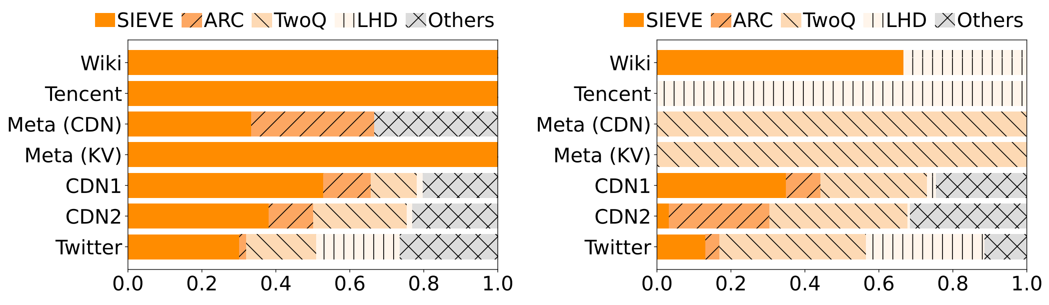


(b) Small cache

Figure 4.4: Miss ratio reduction on Meta (KV + CDN), Wiki CDN, and Tencent Photo CDN datasets. The different opacity of the same color indicates multiple traces from the dataset.

a 1.5% mean miss ratio reduction on the huge number of traces is significant. For example, ARC only reduces LRU's miss ratio by 6.3% on average (not shown). A similar observation can be made on the CDN2 dataset. Although LHD is the best algorithm on the Twitter dataset, Sieve scores second and outperforms most other state-of-the-art algorithms.

When the cache is very small, TwoQ and LHD sometimes outperform Sieve. This is because TwoQ and LHD can quickly remove newly-inserted low-value objects similar to Sieve. The primary reason for Sieve's relatively poor performance is that new objects cannot demonstrate their popularity before being evicted when the cache size is very small. A similar problem also happens with ARC and LIRS. ARC's adaptive algorithm sometimes shrinks the recency queue to very small and yields a high miss ratio. LIRS, which uses a 1% queue for new objects, suffers the most when the cache size is small, as we see its miss ratio on some traces higher than FIFO. In contrast, TwoQ does not suffer from the small cache sizes, because it reserves a fixed 25% of the cache space for new objects, preventing overly aggressive demotion. However, we remark that the production miss ratios reported in previous works [27, 99, 225, 226] are close to the miss ratios we observe at the large cache size.



(a) Large cache **(b)** Small cache

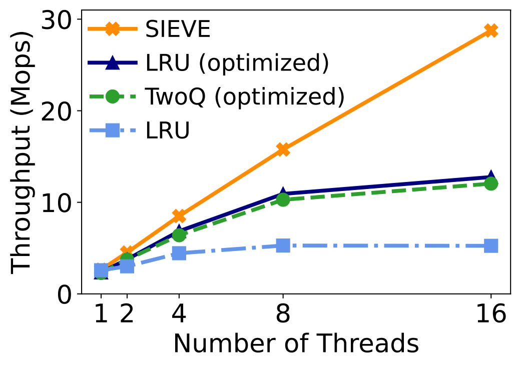
Figure 4.5: Best-performing algorithms on each dataset. Table 4.1 shows the number of traces per dataset.

The secret behind SIEVE's efficiency is the ability to quickly remove newly-inserted unpopular objects (quick demotion), the ability to sift out old unpopular objects, and the balance between new and old objects. We discuss more in §4.4.

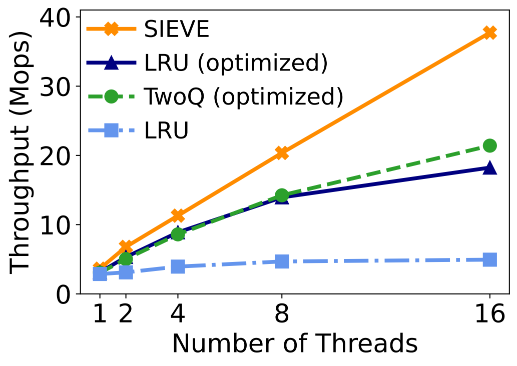
Four small datasets: Meta KV, Meta CDN, Wiki, and TencentPhoto. Because each dataset contains fewer than ten traces, we use scatter plots to compare the algorithms. Fig. 4.4 demonstrates that SIEVE outperforms all other algorithms on all four datasets at the large cache size. When the cache size is small, the observation is similar to that made in Fig. 4.3. SIEVE is the best algorithm on the Wiki dataset. TwoQ and LHD are the best on Meta and TencentPhoto datasets. Although not the best, SIEVE remains highly competitive.

Best-performing algorithm per dataset. We have demonstrated that SIEVE provides larger miss ratio reductions across traces than state-of-the-art algorithms. For a more quantitative comparison, Fig. 4.5 shows the fraction of traces each algorithm performs the best.

With a large cache size, SIEVE outperforms all other algorithms on the Tencent Photo, Wiki, and Meta KV datasets. On the CDN1 and CDN2 datasets, SIEVE is the best algorithm on 48% and 38% of the 1273 and 219 traces. On the Twitter dataset, although SIEVE is the best on only 30% of the traces, it is important to note that no other algorithms are the best on more than 18% of the traces. When using the small



(a) Meta KV trace



(b) Twitter trace

Figure 4.6: Throughput scaling with CPU cores on two KV-cache workloads.

cache size, SIEVE, TwoQ is the best algorithm winning on the two Meta datasets. On the other datasets, SIEVE and LHD have similar shares being the best-performing algorithms. The reason for the observation is similar to that previously explained.

4.3.3 Throughput Performance

Besides efficiency, throughput is the other important metric for caching systems. Although we have implemented SIEVE in five different libraries, we focus on Cachelib's results. Because all other libraries implement strict LRU and do not consider object sizes, evaluations yield the same miss ratio as our simulation. Moreover, strict LRU is not scalable, as we show next.

Fig. 4.6 shows how throughput grows with the number of trace replay threads using two production traces from Meta and Twitter. To better emulate real-world deployments in which the working set size (dataset size) grows with the hardware specs (#cores and DRAM sizes), we scale the cache size and working set size together with the number of threads. To scale the working set size, each thread plays the same trace with the object id transformed into a new space. For example, the benchmark sends 4× more requests to 4× larger cache size at 4 threads compared to the single-thread experiment. We set the cache size to be $4 \times n_{thread}$ GB for both traces, which gives miss ratios of 7% (Meta) and 2% (Twitter). We remark that the miss ratio is close to

Table 4.2: Lines of code requires modification and required engineering time to add SIEVE to a production cache library.

Cache library	Language	Lines
groupcache [44]	Golang	21
mnemonist [6]	Javascript	12
lru-rs [4]	Rust	16
lru-dict [3]	Python + C	21

previous reports [27, 226].

The LRU and TwoQ in Cachelib use extensive optimizations to improve the scalability. For example, objects that were promoted to the head of the queue in the last 60 seconds are not promoted again, which reduces lock contention without compromising the miss ratio. Cachelib further adds a lock combining technique to elide expensive coherence and synchronization operations to boost throughput [64]. As a result of the optimizations, both LRU and TwoQ show impressive scalability results compared to the unoptimized LRU: the throughput is 6× higher at 16 threads than using a single thread on the Twitter trace. As a comparison, unoptimized LRU's throughput plateaus at 4 threads.

Compared to these LRU-based algorithms, SIEVE does not require "promotion" at each cache hit. Therefore, it is faster and more scalable. At a single thread, SIEVE is 16% (17%) faster than the optimized LRU (TwoQ) and on both traces. At 16 threads, SIEVE shows more than 2× higher throughput than the optimized LRU and TwoQ on the Meta trace.

4.3.4 Simplicity

Prototype implementations. SIEVE not only achieves better efficiency, higher throughput, and better scalability, but it is also very simple. We chose the most popular cache libraries/systems from five different languages: C++, Go, JavaScript, Python, and Rust, and replaced the LRU with SIEVE.

Table 4.3: Lines of code (excluding comments and empty lines) and per-object metadata size required to implement each algorithm in our simulator. We assume that frequency counter and timestamps use 4 bytes and pointers use 8 bytes.

Algorithm	cache hit	eviction	insertion	metadata size
FIFO	1	4	3	16B
LRU	5	4	3	16B
ARC	64	108	20	17B
LIRS	96	120	64	17B
LHD	192	81	64	13B
LeCaR	72	76	20	40B
CACHEUS	168	140	150	54B
TwoQ	28	16	8	17B
Hyberbolic	4	20	4	16B
CLOCK	4	9	3	17B
SIEVE	4	9	3	17B

Although different libraries/systems have different implementations of LRU, e.g., most use doubly-linked-list, and some use arrays, we find that implementing SIEVE is very easy. Table 4.2 shows the number of lines (not including the tests) needed to replace LRU — all implementations require no more than 21 lines of code changes. Due to many locks and optimizations in Cachelib are no longer needed for SIEVE, quantifying the code modifications is impossible. Therefore, the Cachelib implementation is not included in Table 2.

Advanced algorithms in simulator. Because most of the complex algorithms we evaluated in §4.3.2 are not implemented in production systems. Therefore, we compare the lines of code needed to implement cache hit, insert, and evict in our simulator. Although we implemented a linked list and hash table in our simulator in C, we do not include the lines related to list and hash table operations, i.e., appending to the list head or inserting to the hash table requires one line.

Table 4.3 shows that FIFO requires the fewest number of lines to implement. On top of FIFO, implementing LRU adds a few lines to promote an object upon cache hits. CLOCK and SIEVE require close to 10 lines to implement the eviction function because both need to find the first object that has not been visited. However, we remark that SIEVE is simpler than LRU and CLOCK because SIEVE does not require moving objects to the head of the queue in either hit or miss (evict). Besides these, all other algorithms require one to two orders more lines of code to implement the three functions.

Per-object metadata. In addition to the implementation complexity, we also quantified the per-object metadata needed to implement each algorithm. FIFO does not require any metadata when implemented using a ring buffer. However, such an implementation does not support overwrite or delete. So common FIFO implementation also uses a doubly-linked list with 16 bytes of per-object metadata similar to LRU. CLOCK and SIEVE are similar, both requiring 1-bit to track object access status. When implemented using a doubly linked list, they use 17 bytes per-object metadata. Compared to SIEVE, advanced algorithms often require more per-object metadata. Many key-value cache workloads have objects as small as 10s of bytes [138, 225], and a large metadata wastes the precious cache space.

ZERO parameter. *Besides being easy to implement and having less metadata,* SIEVE *also has no parameters.* Except for FIFO, LRU, CLOCK, and Hyperbolic, all other algorithms have explicit or implicit parameters, e.g., the sizes of queues in LIRS, the learning rate in LeCaR and CACHEUS, the decay rate and age granularity in LHD. Note that although ARC has no explicit parameters, its adaptive algorithm uses implicit parameters in deciding when and how much space to move between the queues. As a comparison, SIEVE has no parameter and requires no tuning.

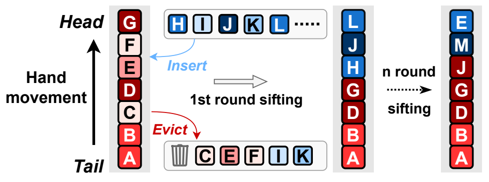
4.4 Distilling SIEVE's Effectiveness

Our empirical evaluation shows that Sieve is simultaneously simple, fast, scalable, and efficient. In a well-trodden field like cache eviction, Sieve's competitive performance was a genuine surprise to us as well. We next report our analysis that seeks to understand the secrets behind its efficiency.

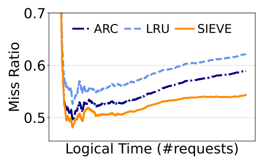
4.4.1 Visualizing the Sifting Process

The workhorse of Sieve is the "hand" that functions as a sieve: it sifts through the cache to filter out unpopular objects and retain the popular ones. We illustrate this process in Fig. 4.7a, where each column (queue) represents a snapshot of the cached objects over time from left to right. As the hand moves from the tail (the oldest object) to the head (the newest object), objects that have not been visited are evicted – the same sweeping mechanism that underlies CLOCK [50, 59]. For example, after the first round of sifting, objects at least as popular as A remain in the cache while others are evicted. The newly admitted objects are placed at the head of the queue — much like the CLOCK policy, but a departure from CLOCK, which does in-place replacements to emulate LRU. During the subsequent rounds of sifting, if objects that survived previous rounds remain popular, they will stay in the cache. In such a case, since most old objects are not evicted, the eviction hand quickly moves past the old popular objects to the queue positions close to the head. This allows newly inserted objects to be quickly assessed and evicted, putting greater eviction pressure on unpopular items (such as "one-hit wonders") than LRU and its variations [139]. As previous work has shown [33, 99, 220], quick demotion is crucial for achieving high cache efficiency.

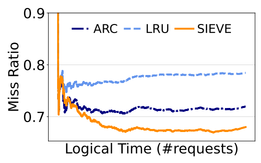
Fig. 4.7b and Fig. 4.7c show the cumulative miss ratio over time of different algorithms on two representative production traces. After the cache is warmed up,



(a) Density of colors indicates inherent object popularity (blue: newly inserted objects; red: old objects in each round), and the letters represent object IDs. The first queue captures the state at the start of the first round, and the second queue captures the state at the end of the first round.



(b) Trace 1, full trace (one week)



(c) Trace 2, first two days of a week-long trace

Figure 4.7: Left: illustration of the sifting process. Right: Miss ratio over time for two traces. The gaps between SIEVE's miss ratio and others enlarge over time.

the miss ratio gaps between SIEVE and other algorithms widen over time, supporting the interpretation that SIEVE indeed sifts out unpopular objects and retains popular ones. A similar observation can be seen in Fig. 4.10a.

4.4.2 Analyzing the Sifting Process

We now analyze the popularity retention mechanism in SIEVE. To clarify the exposition, suppose the SIEVE cache can fit C equally sized objects. Since SIEVE always inserts new objects at the head, and objects that are retained remain in their original positions within the queue, the algorithm implicitly partitions the cache between new and old objects. This partition is dynamic, allowing SIEVE to strike a balance between exploration (finding new popular objects) and exploitation (enjoying hits on old popular objects).

SIEVE performs sifting by moving the hand from the tail to the head, evicting unpopular objects along the way, which we call one round of sifting. We use r to denote the number of rounds. We first enumerate the queue positions p from the tail ($p = 0$) to the head ($p = C - 1$). We then further denote that an object at position p in round r is *examined* (during eviction) or *inserted* at time T_p^r. Note that

T effectively defines a logical timer for the examined objects: whenever an object is examined, T increases by 1, regardless of whether the examined object is evicted or retained. In addition, T changes *once* each round for an old object (retained from previous rounds).

For an old object x at position p, we define the "inter-examination time" $I_e(p^r) = T_p^r - T_{p'}^{r-1}$ where p' was the position of x in round $r-1$. Clearly, $p' \geq p$. For a new object inserted in the current round, the inter-examination time is defined as the time between its examination and insertion. We further define an old object x's "inter-arrival time" $I_a(x^r)$ as the time, measured again in the number of objects examined, between the first request to the x in round r and the last request to x in round $r-1$. For a new object, the inter-arrival time is the time between its insertion and the second request. If an old object is not requested in the last round or a new object does not have a second request, its inter-arrival time is infinite.

In round r, consider two consecutive retained objects x_1 and x_2 at position p_1 and $p_2 = p_1 + 1$. The inter-examination times are $I_e(p_1^r) = T_{p_1}^r - T_{p_1'}^{r-1}$ and $I_e(p_2^r) = T_{p_2}^r - T_{p_2'}^{r-1}$, respectively. The transition yields two invariants:

$$\begin{aligned} T_{p_2}^r - T_{p_1}^r &= 1 \\ T_{p_2'}^{r-1} - T_{p_1'}^{r-1} &\geq 1 \end{aligned}$$

The first equation follows from x_1 and x_2 being consecutively retained objects; the second inequality expresses that other evictions may have taken place between x_1 and x_2 in the previous round. Together, these imply that $I_e(p_1^r) \geq I_e(p_2^r)$. The result generalizes further: for any two retained old objects in the queue, the object closer to the head has a smaller inter-examination time.

Moreover, if an object is retained, its inter-arrival time must be no greater than its inter-examination time. Therefore, for any retained object x at position p_x, its inter-arrival time $I_a(x^r)$ must be smaller than the tail object's inter-examination time:

$$I_a(x^r) \leq I_e(p_x^r) \leq I_e(p_0^r) \tag{4.1}$$

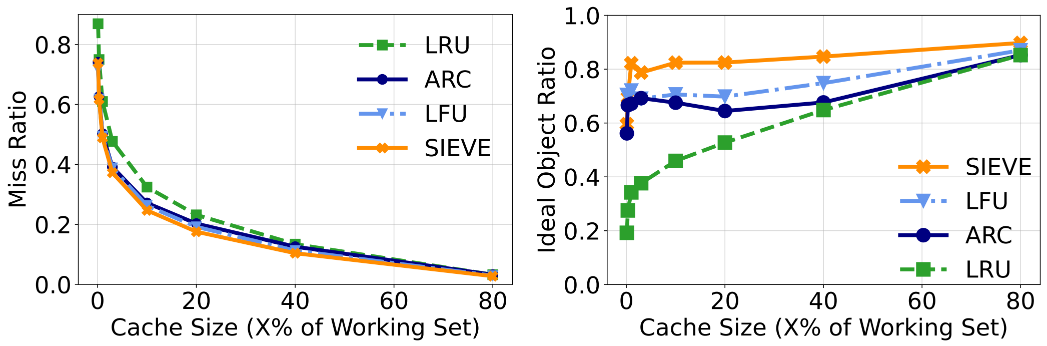
Using the commonly assumed independent reference model [51, 82, 100, 101] with a Poisson arrival, we can expect any retained object to be more popular than some dynamic threshold set by the tail object's inter-examination time $I_e(p_0^r)$. Since evicting an object keeps the hand pointer at its original position (relative to the tail), the more objects are evicted during a round, the longer the inter-examination time. As a result, SIEVE effectively adapts the popularity threshold so that more objects are retained in the next round.

Following our sifting process metaphor, the mesh size in SIEVE is determined by the tail object's inter-examination time $I_e(p_0^r)$, which is dynamically adjusted based on object popularity change. If too few objects are retained in one round (mesh size too small), then we will have an increased tail inter-examination time $I_e(p_0^r)$ (a larger mesh size) in the next round.

4.4.3 Deeper Study with Synthetic Workloads

Production trace workloads are often too complex and dynamic to analyze. One consistent finding from past workload characterization work, however, is that object popularity in web cache workloads invariably follows a heavy-tailed power-law (generalized Zipfian) distribution [46, 225]. Therefore, we employed synthetic power-law workloads for our study. They replicate these real-world patterns in a controlled manner, allowing for more focused and repeatable experimentation. Using these, we further scrutinize SIEVE's effectiveness.

Miss ratio over size. Fig. 4.8a displays the miss ratio of LRU, LFU, ARC, and SIEVE at different cache sizes. Notably, LFU, ARC, and SIEVE all exhibit lower miss

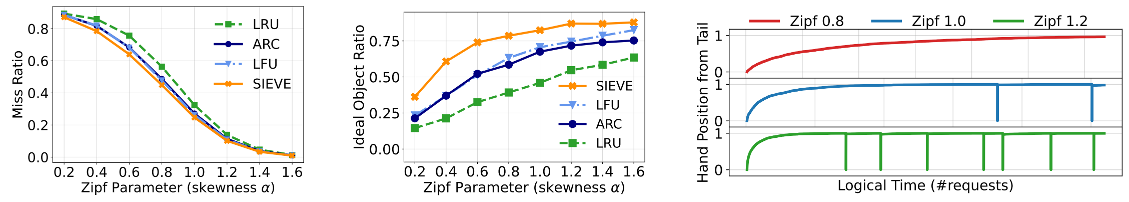


(a) Miss ratio over size **(b)** Ideal object ratio over size

Figure 4.8: Miss ratio and ideal object ratio on a Zipfian dataset ($\alpha = 1.0$).

ratios than LRU, demonstrating their efficiency. Despite being considered optimal for synthetic power-law workloads, LFU performs similarly to ARC and is visibly worse than Sieve. This is because objects with medium popularity, such as objects with ranks around the cache size C, are only requested once before their eviction. LFU cannot distinguish the true popularity of these objects and misses out on an opportunity for better performance. As a comparison, both ARC and Sieve can quickly remove new and potentially unpopular objects, which allows cached objects to enjoy more time in the cache to demonstrate their popularity. Between the two algorithms, Sieve further extends the tenure of these objects in the cache because when the hand sweeps through the newly inserted objects, the objects closer to the head must have strictly shorter inter-arrival times (expected to be more popular) to survive.

Ideal object ratio over size. To capture how different algorithms manage popular objects, we define a metric called "ideal object ratio". Under the assumption of a static and known popularity distribution, the optimal caching policy retains the most popular content within the cache at all times. Given a cache size C and a workload following a power-law distribution, the ideal objects are the C most frequent objects in the workload, denoted by H. The ideal ratio of objects in the cache at time t is calculated by $I_t = \frac{|H \cap A_t|}{C}$ where A_t denotes the cache contents at time t.



(a) Miss ratio **(b)** Ideal object ratio **(c)** Hand movement in SIEVE

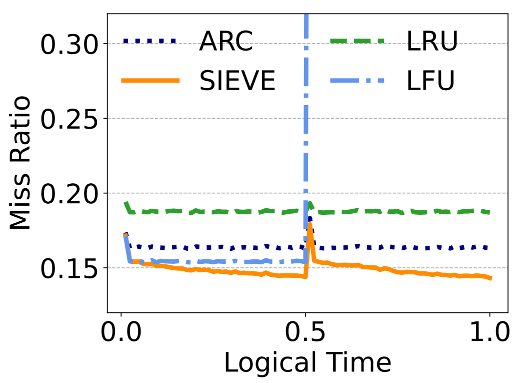
Figure 4.9: Left two: miss ratio and ideal object ratio on Zipfian workloads with different α. Right: hand position in the cache over time in Zipfian workloads.

Fig. 4.8b shows the ideal object ratio at different cache sizes. LRU evicts objects based on recency, which only weakly correlates with popularity. In this scenario, LRU stores the least number of popular objects. LFU stores slightly more "ideal objects" than ARC. SIEVE, however, successfully filters out unpopular objects from the cache.

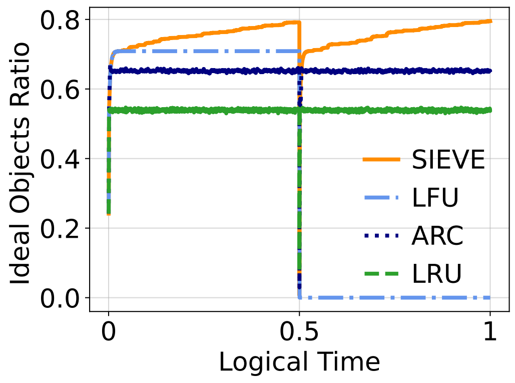
Varying the popularity skew. Fig. 4.8 shows a distribution with Zipfian skewness $\alpha = 1$. We further studied how different concentration of popularity affects SIEVE's effectiveness. Due to space restrictions, we focus on results with large cache sizes for the remainder of this subsection. Results using the small cache size are either similar or do not reveal interesting patterns.

Fig. 4.9a and Fig. 4.9b the impact of varying skew on miss and ideal object ratios. As skew increases, making popular objects more prominent, it becomes easier to identify and cache the ideal objects, increasing the ideal object ratio and reducing the miss ratio for all tested algorithms. Among ARC, LFU, and SIEVE, we observe that SIEVE always shows a higher ideal ratio with a lower miss ratio across skewness, indicating the efficiency of SIEVE is not limited to very skewed workloads.

Fig. 4.9c illustrates the hand position in the SIEVE cache over time, advancing towards the head with each retained object and pausing during evictions. Therefore, the more objects are retained, the faster the movement. We observe that the hand moves more slowly in the first round than in the later rounds because that is when many unpopular objects are evicted. In subsequent rounds, the hand lingers at positions close to the head for most of the time because SIEVE keeps a new object



(a) Interval miss ratio



(b) Ideal object ratio over time

Figure 4.10: Interval miss ratio and ideal object ratio over time on a workload constructed by connecting two different Zipfian workloads ($\alpha = 1$).

at position p only if it is more popular (shorter inter-arrival time) than the object at position $p - 1$. In other words, SIEVE performs quick demotion [210].

In more skewed workloads, the hand moves quickly due to early arrival and higher request volumes for popular objects, allowing SIEVE to cache most ideal objects by the end of the first round. Consequently, the hand rapidly transitions from tail to head with fewer evictions and spends less time near the head, as new objects are more likely to be retained, hastening its progress. Nevertheless, the time of each round varies depending on the frequency of encountering potentially popular objects, highlighting SIEVE's adaptability to workload shifts. When new popular objects appear, the hand accelerates, replacing existing cached objects with the newcomers by giving less time to set their visited bit.

Sieve is adaptive. To visualize SIEVE's adaptivity via the sifting process, we created a new workload by joining two Zipfian ($\alpha = 1.0$) workloads that request different populations of objects. Fig. 4.10 shows the interval miss ratio (per 100,000 requests) over time on this conjoined workload. The changeover happens at the 50% midway time mark. We observe that the interval miss ratio of LFU skyrockets to nearly 100% (beyond figure bounds) since new objects cannot replace the old objects. Relative to LRU and ARC, SIEVE's miss ratio spike is larger because it takes time for the hand to move back to the tail before it can evict old objects. However, Sieve's spike is invisible when the cache size is small (not shown). With respect to the interval miss ratio spike, we observe the ideal object ratio of all algorithms (the curves overlap) dropping to 0 when the workload changes at the midway point. Whereas LFU never recovers from the drop, the ideal miss ratios in all other algorithms quickly recover to large proportions. Finally, the figures corroborate our interpretation of the sifting process: Sieve's miss ratio drops over time, while the fraction of ideal objects increases over time.

4.5 SIEVE as a Turn-key Cache Primitive

4.5.1 Eviction Algorithm Designs

Beyond being a cache eviction algorithm, SIEVE can serve as a cache primitive for designing more advanced eviction policies. To study the range of such policies, we categorize existing cache eviction algorithm designs into four main approaches. **(1)** We can design simple and easy-to-understand eviction algorithms, such as FIFO queues, LRU queues, LFU queues, and Random eviction. We call these simple algorithms *cache primitives*. SIEVE falls under this category. **(2)** We can improve the cache primitives. For example, FIFO-Reinsertion is designed by adding reinsertion to FIFO; LRU-K [150] is designed by changing the recency metric in LRU. **(3)** We can compose multiple cache primitives with objects moved between them. For example, ARC, SLRU, and MQ use multiple LRU queues. **(4)** We can run multiple cache primitives and craft a decision-maker to select eviction candidates suggested by the primitives. For example, LeCaR [202] uses reinforcement learning to choose between the eviction candidates from LRU and LFU; HALP [183] uses machine learning (MLP) to choose one object from the eight objects at the LRU tail.

Having an efficient cache primitive not only provides an effective and simple eviction algorithm but also enables other approaches to design more efficient algorithms.

4.5.2 Efficient Cache Primitives

The ideal cache primitive is simultaneously (1) simple, (2) efficient, and (3) fast — in terms of high throughput. For example, FIFO and LRU meet these requirements and are frequently used to construct more advanced algorithms. However, they are less efficient than complex algorithms.

Recent work shows that lazy promotion and quick demotion of cache objects are key to effective cache eviction processes [220]. Unlike existing cache primitives, which

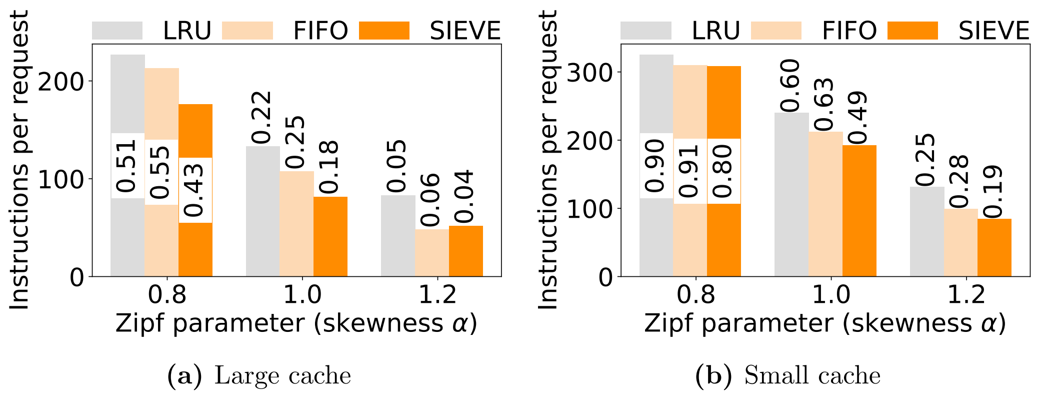
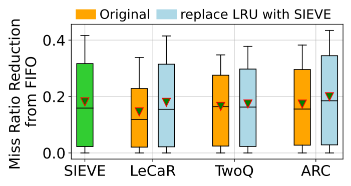


Figure 4.11: Average number of instructions per request when running LRU, FIFO, and SIEVE caches. The top number denotes the miss ratio.

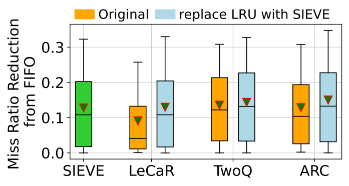
do not support both properties, SIEVE *is the first cache primitive that supports both lazy promotion and quick demotion*. This serves as the foundation for SIEVE's high efficiency and high performance.

While we have shown that SIEVE is simple, efficient, and fast in §4.3, to further understand SIEVE as a cache primitive, we compare the number of instructions needed to run FIFO, LRU, and SIEVE caches. We remark that the number of instructions may not necessarily correlate with latency or throughput but rather a rough metric of CPU resource usage. We used `perf stat` to measure the number of instructions for serving power-law workloads (100 million requests, 1 million objects) in our simulator. We then deduct the simulator overhead by measuring a no-op cache, which performs nothing on cache hits and misses.

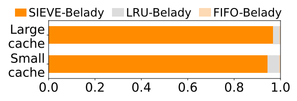
Fig. 4.11 shows that SIEVE generally executes fewer instructions per request than FIFO and LRU, a difference accentuated in skewed workloads and larger cache sizes. Compared to LRU, SIEVE requires fewer instructions since SIEVE needs only to check and possibly update a Boolean field on cache hits, which is much simpler than moving an object to the head of the queue. Besides LRU, SIEVE also requires fewer instructions than FIFO because of the difference in miss ratios. Because SIEVE has a lower miss ratio than FIFO, fewer objects need to be inserted due to cache misses, leading to fewer instructions per request on average. The only exception is when SIEVE and



(a) Large cache



(b) Small cache



(c) Best-performing algorithms across traces.

Figure 4.12: Impact of replacing LRU with Sieve in advanced algorithms (**a**,**b**). The potential of FIFO, LRU, and Sieve when endowed with foresight (**c**).

FIFO have similar miss ratios, in which case, FIFO executes fewer instructions than Sieve. Overall, Sieve requires up to 40% and 24% fewer instructions than LRU and FIFO, respectively.

4.5.3 Turn-key Cache Eviction with Sieve

As a cache primitive, Sieve can facilitates the design of more advanced eviction algorithms. To understand the benefits of using a better cache primitive, we replaced the LRU in LeCaR, TwoQ, and ARC with Sieve. Note that for ARC, we only replace the LRU for frequent objects.

We evaluate these algorithms on the 1559 traces[4] and show the miss ratio reduction(from FIFO) in Fig. 4.12a and Fig. 4.12b. Compared to Sieve, LeCaR has much lower efficiency; however, when replacing the LRU in LeCaR with Sieve, it significantly reduces LeCaR's miss ratio by 4.5% on average. TwoQ and ARC achieve efficiency close to Sieve; however, when replacing the LRU with Sieve, the efficiency of both algorithms gets boosted. For example, ARC-Sieve achieves the best efficiency among all compared algorithms at both small and large cache sizes. Specifically, it reduces ARC's miss ratio by 3.7% on average and up to 62.5% on the large size (recall that ARC reduces LRU's miss ratio by 6.3% on average). Compared to Sieve, ARC-Sieve reduces miss ratio by 2.4% on average and up to 40.6%.

[4]We do not show each dataset separately to save space

To understand the potential in suggesting eviction candidates, we evaluated the efficiency of FIFO, LRU, and SIEVE, granting them access to future request data. Each eviction candidate is either evicted or reinserted, depending on whether the object will be requested soon. We assume that an object will be requested soon if the logical time (number of requests) till the object's next access is no more than $\frac{C}{mr}$, where C is the cache size and mr is the miss ratio. This mimics the case that we have a perfect decision-maker choosing between the eviction candidates suggested by multiple simple eviction algorithms. Fig. 4.12c shows that when supplied with this additional information, SIEVE achieves the lowest miss ratio on 97% and 94% of the 1559 traces at the large and small cache size, respectively.

These results highlight the potential of SIEVE as a powerful cache primitive for designing advanced cache eviction algorithms. Leveraging lazy promotion and quick demotion, SIEVE not only performs well on its own but also bolsters the performance of more complex algorithms.

4.6 Discussion

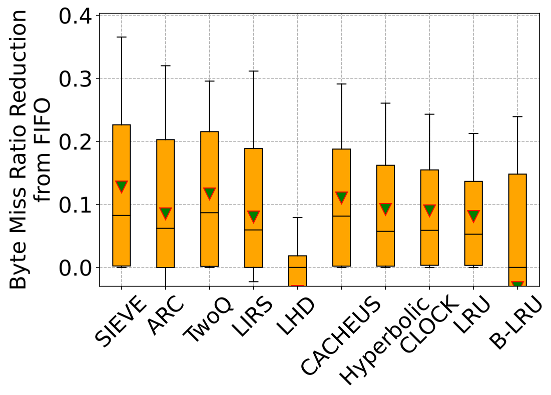
4.6.1 Byte Miss Ratio

To gauge SIEVE's efficiency in reducing network bandwidth usage in CDNs, we analyzed its byte miss ratio by considering object sizes. We chose the cache size at 10% and 0.1% of the trace footprint in bytes. Fig. 4.13a and Fig. 4.13b show that SIEVE presents larger byte miss ratio reductions at *ALL* percentiles than state-of-the-art algorithms at both cache sizes, showcasing its high efficiency in CDN caches.

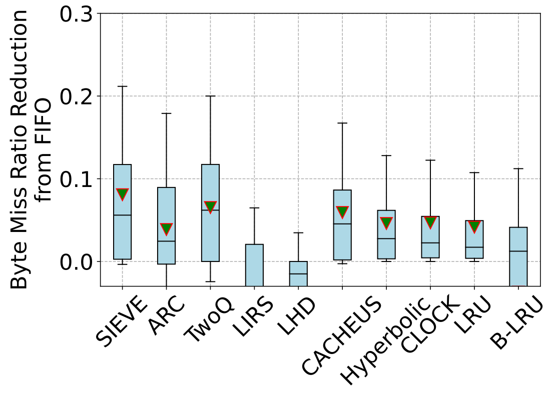
We further compared SIEVE with LRB [182], the state-of-the-art machine-learning-based cache eviction algorithm optimized for byte miss ratio. Due to LRB's long run time, we only evaluated LRB on the two open-source Wiki traces provided by the authors. Fig. 4.14a and Fig. 4.14b show that LRB performs better at small cache sizes (1% and 2%), while SIEVE excels at larger cache sizes. We conjecture that at a small cache size, the ideal objects to cache are popular objects with many requests, which LRB can more easily identify because they have more features (most of LRB's features are about the time between accesses to an object). When the cache size is large, most objects in the cache have few requests. Without enough features, a learned model can provide little benefits [220, 227]. In summary, compared to complex machine-learning-based algorithms, SIEVE still has competitive efficiency.

4.6.2 Sieve is Not Scan-resistant

Besides web cache workloads, we evaluated SIEVE on some block cache workloads. However, we find that SIEVE sometimes shows a miss ratio higher than LRU. The primary reason for this discrepancy is that SIEVE is not scan-resistant. In block cache workloads, which frequently feature scans, popular objects often intermingle with objects from scans. Consequently, both types of objects are rapidly evicted after insertion. Since SIEVE does not use a ghost cache, it cannot recognize the popular

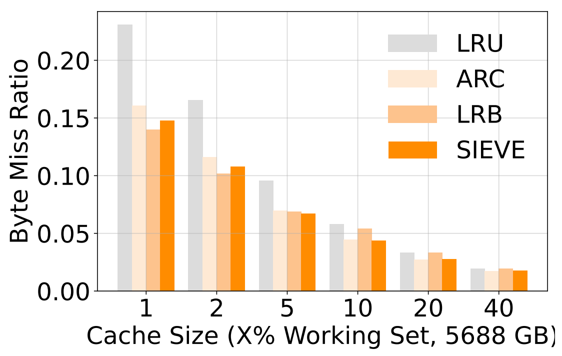


(a) Large cache

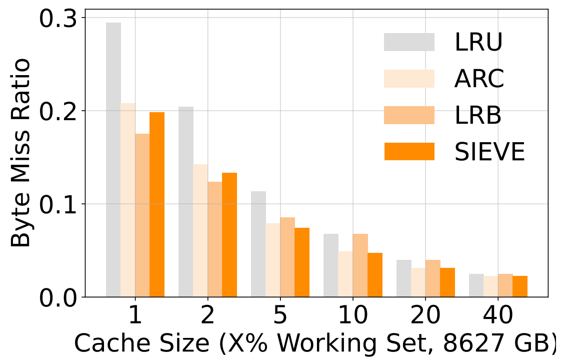


(b) Small cache

Figure 4.13: Byte miss ratio across all CDN traces.



(a) Wiki2018 trace



(b) Wiki2019 trace

Figure 4.14: Byte miss ratios at different cache sizes on two Wiki CDN traces used in LRB evaluation.

objects when they are requested again. This problem is less severe on the large cache size, but when the cache size is small, we observe that having a ghost is critical to be scan-resistant. We conjecture that not being scan-resistant is probably the reason why SIEVE remained undiscovered over the decades of caching research, which has been mostly focused on page and block accesses.

4.6.3 TTL-friendliness

Time-to-live (TTL) is a common feature in web caching [225, 226]. It specifies the duration during which an object can be used. After the TTL has elapsed, the object expires and can no longer be served to the user, even if it may still be cached. Most existing eviction algorithms today do not consider object expiration and require a separate procedure, e.g., scanning the cache, to remove expired objects. Similar to FIFO, Sieve maintains objects in insertion order, which allows objects in TTL-partitioned caches, e.g., Segcache [226], to be sorted by expiration time. This provides a convenient method for discovering and removing expired objects.

4.7 Conclusion

We design Sieve, a simple, efficient, fast, and scalable cache eviction algorithm for web caches that leverages “lazy promotion” and “quick demotio”. The high efficiency in SIEVE comes from gradually sifting out the unpopular objects. SIEVE is the first and the simplest cache primitive that supports both lazy promotion and quick demotion. This serves as the foundation for SIEVE’s high efficiency and high performance. Evaluated on 1559 traces from 7 datasets, we show that SIEVE outperforms complex state-of-the-art algorithms on over 45% of the traces. We implemented SIEVE in five open-source production libraries using less than 20 lines on average.

Chapter 5

Theodon: A Modular Framework for CDN Optimization

In this chapter, we introduce Theodon, a framework designed to efficiently identify CDN configurations that optimize both performance and cost. Theodon employs modular components to simulate complex CDN topologies, enabling efficient discovery of configurations that achieve an effective balance between performance and cost. Additionally, this chapter explores the application of Theodon in enhancing real CDN systems like Cloudflare and WikimediaCDN, demonstrating its practical utility in optimizing network performance.

5.1 Introduction

A Content Delivery Network (CDN) is a large, globally distributed system comprising hundreds of thousands of servers. CDNs are crucial for a variety of internet services, from streaming and social media to extensive web applications, playing a vital role in delivering a smooth and efficient user experience. For example, CDN caching handles 70% of web requests for companies like Meta [99]. To minimize latency, CDNs strategically position edge servers in multiple locations, bringing content closer to end-users. This can significantly reduce latency, for instance, from 150 milliseconds to 30 milliseconds [99, 137, 148, 241].

However, the complexity of CDN operations presents substantial challenges. Despite their extensive deployment, finding the optimal configuration that balances objectives like latency, throughput, and cost remains elusive. This is often due to reliance on empirical methods, short-term evaluations, or anecdotal evidence when deciding on crucial operational parameters, such as cache sizes, data center locations, and CDN tiering. These methods fall short in providing a systematic, long-term perspective on factors impacting CDN performance.

To address this, we introduce THEODON, a framework that models CDN architectures through modular simulations. It aims to learn near-optimal configurations that balance performance and cost objectives. THEODON views a CDN as a composable system, constructed from predefined components like Points of Presence (PoPs), DRAMs, SSDs, load balancers, eviction algorithms, and etc. Each component features adjustable parameters; for instance, a PoP can be configured by its size, machine count, and geographic location. A configuration, therefore, is the amalgamation of all component parameters, defining a specific CDN system setup. THEODON employs multiple objective Bayesian Optimization to explore this parameter space, seeking near-optimal configurations that strike an optimal balance between performance and cost.

In our study, we simulate the real-world WikimediaCDN and Cloudflare networks, evaluating THEODON using traces from WikimediaCDN. Our results demonstrate that THEODON is effective in identifying configurations that enhance performance by up to 10% and 23% compared to the default settings of WikimediaCDN and Cloudflare, respectively. Additionally, we assess THEODON's ability to balance the trade-off between performance and cost. Notably, while maintaining a byte miss ratio similar to Cloudflare's default setting, THEODON can efficiently pinpoint a configuration that significantly reduces costs, achieving approximately 2.4 times lower expenses.

5.2 Motivation

Content Delivery Networks (CDNs) are instrumental in managing a significant portion of global internet traffic. These vast networks, composed of hundreds of PoPs worldwide, are essential yet complex systems. Despite their importance, CDN optimization often relies on heuristic methods developed over many years. This reliance poses challenges due to the nonlinear and unpredictable nature of CDNs, essentially large-scale cache systems. A systematic approach to evaluate the trade-offs in topology, parameter settings, and resource allocation is currently lacking.

5.2.1 CDN Examples.

This subsection delves into the intricacies of Content Delivery Networks (CDNs), using two examples - the Wikimedia CDN and Cloudflare. By examining these real-world applications, we aim to illustrate the diverse strategies and configurations deployed in CDNs, highlighting their implications for performance and cost-efficiency.

The Wikimedia Foundation, a top global web entity, operates its own CDN to handle a vast majority of its web traffic. The foundation's CDN architecture primarily relies on two main data centers and four cache-only Points of Presence (PoPs) [169]. A recent PoP, set up in 2022, consists of 16 dedicated servers, with equal distribution for processing text and image web requests. These servers are each equipped with 384GB of RAM and 1.6TB of NVMe SSD storage [2, 11]. The request flow within a PoP is depicted in Fig. 5.1: upon receiving a request, the load balancer uses consistent hashing on the client's IP to select a cache server. If there's a cache hit in the DRAM cache, the response is sent directly back to the user. In the case of a cache miss, the DRAM cache uses consistent hashing on the request URL to redirect the request to the SSD cache within the same PoP, and if necessary, to the next PoP or the origin server.

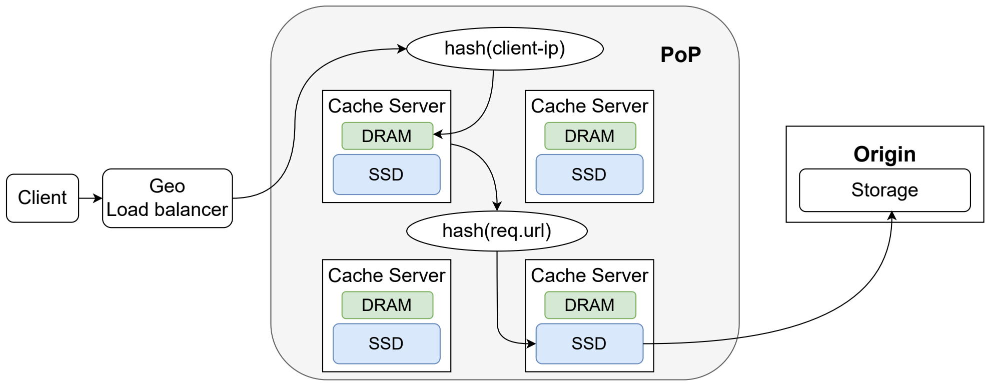


Figure 5.1: A general topology of a CDN PoP.

Cloudflare represents another exemplary CDN, known for handling millions of cache hits per second globally. Similar to Wikimedia, Cloudflare's CDN topology involves multiple servers with DRAMs and SSDs, utilizing consistent hashing for efficient request routing. However, Cloudflare distinguishes itself with its deployment scale and its innovative approach to caching. Cloudflare's infrastructure is also known for its resilience and security features, making it a preferred choice for many organizations seeking robust content delivery solutions. However, this aspect falls beyond the scope of our current study.

Both CDNs employ a similar infrastructure of servers with DRAMs and SSDs, along with consistent hashing for request routing. However, they differ in their caching approaches. Wikimedia CDN uses an LRU (Least Recently Used) algorithm for its in-memory cache and FIFO (First In, First Out) for on-disk cache, whereas Cloudflare applies LRU for both in-memory and SSD caches. Furthermore, Wikimedia CDN prioritizes keeping popular objects in DRAM, contrasting with Cloudflare's strategy of placing less popular objects in DRAM to minimize disk writes and extend SSD lifespans. These differences underscore the versatility in CDN configurations and the importance of tailoring strategies to specific operational requirements and traffic patterns.

5.2.2 Challenges

Complex configuration space. As discussed in §2.3, CDN is a complicated system, influenced by numerous factors. Central to these is hardware configuration, which lays the groundwork for the CDN's operational capacity. Equally crucial are the request routing strategies, determining the efficacy and speed of content delivery to end-users. Tiering, involving the structuring of the CDN for optimized resource allocation and response times, is another key aspect. The capacity planning at various Points of Presence (PoPs) also plays a significant role, encompassing decisions about server numbers and the distribution of DRAM and SSD resources to manage diverse loads and data types. Additionally, data movement strategies are critical, covering both the eviction algorithms for data storage management and approaches for handling frequently demanded hot objects. Lastly, the overall network conditions, such as bandwidth capacity, are vital components that can significantly influence CDN performance. The interaction of these factors underscores the complexity of managing and optimizing CDN systems effectively.

Each of the category has massive of parameters to tune. In this work, our focus is primarily on the analysis at the PoP level. We do not delve into the aspects of the placement of different PoPs or the intricacies of network configurations. Table 5.1 shows a detailed overview of the parameters we're examining.

Performance-cost trade-off. While configurations with larger capacity and the fastest storage generally provide better performance, optimizing for running cost is much more difficult. Fig. 5.2 illustrates this dilemma, showcasing the trade-off between byte miss ratio and cost. This is demonstrated through a simulation of about 30,000 different configurations run on a Cloudflare-like topology, under a consistent workload (further details of the simulation will be discussed in section §5.3.1). Unless otherwise specified, we refer to cost as the normalized cost of configured DRAMs and SSDs within a PoP. An interesting observation from our study is the emergence

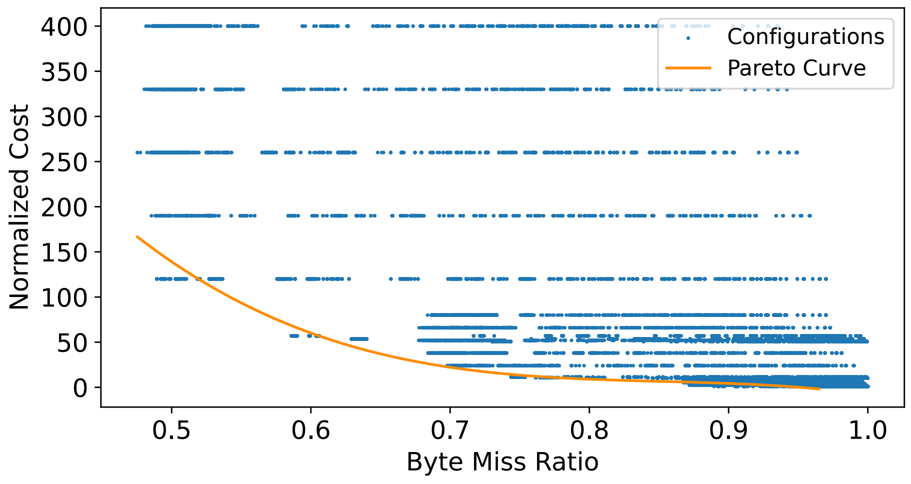


Figure 5.2: Trade-off curve and sub-optimal configurations for tuning CDNs.

of an optimal Pareto curve delineating the performance-cost trade-off. Beyond this curve, improvements in byte miss ratio or cost reductions invariably compromise the other metric. Given the diverse objectives of system operators regarding performance and cost, pinpointing the ideal configuration that satisfies specific requirements is challenging.

5.3 Design

We introduce Theodon, a data-driven framework that leverages modular simulation and Bayesian Optimization to build performance models for CDNs. Theodon identifies best configurations to provide high performance while minimizing cost. Theodon addresses the challenges outlined in §5.2.2 by applying the following approaches:

1. **Modular architecture:** Theodon conceptualizes the CDN as a series of interlinked components, including Tier, PoP, DRAM, SSD, various eviction algorithms, and etc. This modular setup facilitates the assembly of diverse CDN topologies, simulating real-world CDNs as detailed in §5.3.1.

2. **Multi-objective Bayesian Optimization:** Employing qNEHVI [60], a novel multi-objective bayesian optimization approach, Theodon effectively identifies the optimal balance between performance and cost, as described in §5.3.2.

5.3.1 Modular Simulation

Theodon's architecture is built around a set of interlinked components, each representing a fundamental element of CDN infrastructure. This design allows for the assembly of diverse CDN topologies, offering flexibility and precision in simulating different operational scenarios.

Components

Theodon's structure decomposes the complexity of CDNs into manageable components, each representing fundamental element of CDN functionality, as detailed in Table 5.1. The components identified for modular simulation within Theodon have been synthesized from an extensive analysis of several leading commercial CDNs,

Table 5.1: THEODON simulator's components and parameters.

Component	Parameters	Description
Tier	number of tiers	Represents different levels of caching hierarchy in the CDN.
	capacity per tier	
PoP	number of PoPs	Geographical locations where servers are placed for content delivery.
	capacity per PoP	
	number of servers per PoP	
Server	number of DRAM/SSD/HDD	Backbone of the CDN, where content is stored and served.
	size of DRAM/SSD/HDD	
Eviction Algorithm	LRU	Determines how cached content is replaced, impact cache hit rates and efficiency, especially the offload to origin.
	FIFO	
	CLOCK	
	SIEVE	
Load Balancer	Random	Determines network traffic across servers
	URL_Hash	
Dataflow Controller	storage rules for hot objects	Manages content storage based on popularity
	hit requires for promotion	

encompassing MetaPhoto CDN [99], TencentPhoto CDN [241], YouTube CDN [184], Akamai [133, 148], Cloudflare [13], and Wikimedia CDN [169].

- **Tier:** The tier structure reflects the layered hierarchy seen in these CDNs, where each tier can consist of PoPs. Commonly, a cache miss at a lower tier triggers a retrieval from an upper tier, involving a more distant PoP. While this tiered system reduces the frequency of requests reverting to the origin server for identical content—thus enhancing content delivery efficiency—it also entails higher operational costs for the CDN provider.

- **PoP:** Within each tier, there can be several PoPs. These are strategically placed data centers that facilitate content delivery, essential for ensuring content is served to end-users with minimal delay.

- **Server:** Each PoP houses a collection of servers, which are the backbone of a CDN. These servers are equipped with a varied number of DRAM and SSD units to store and serve content.

- **DRAM/SSD:** Nested within servers, these storage components are critical for data access speed and volume. Each server may contain a varying number of

DRAM and SSD units.

- **Eviction algorithm:** This is an adaptable component associated with both DRAM and SSD storage within servers. It dictates the protocol for content replacement, significantly impacting cache efficiency and CDN performance.
- **Load balancer:** Operating as the interconnector within the CDN, the load balancer is the mechanism that routes traffic among servers or between different PoPs, crucial for maintaining optimal load distribution and network performance.
- **Dataflow Controller:** This component oversees the strategic movement of hot objects – frequently requested content – between DRAM and SSD storage mediums, ensuring that popular content is stored and served efficiently according to user demand.

The modularity of the simulator provides flexibility in constructing tailored CDN configurations that accurately simulate real-world setups.

Interface

THEODON exposes the components configurations with a configuration file, which can be used to specify the setup and the behavior of the simulated CDN environment. We describe these configurations and how they map to THEODON's simulator components.

System-wide configurations: The `system_wide` configurations establishes the fundamental environment for the simulation. For example, THEODON simulator enables user to run simulations in parallel (using the `parallel_jobs` parameter), optimizing runtime efficiency.

Workload configurations: Parameters such as `max_iter` and `time_unit` defines the duration and granularity of input traces. Additional settings like `warmup_epoch`, `epoch_type`, and `epoch_unit` provide dynamic and realistic workload modeling.

CDN cache configurations: These configurations can be classified into two categories:

1. *hierarchical CDN structure:* The configuration of tiers (`tier`) and Points of Presence (`pops`) includes detailed settings for load balancers (e.g., `tier_lb`, `pop_lb`), serialization (`tier_serializer`, `pop_serializer`), and data movers (`data_mover` with an enabled switch).

2. *Server configurations:* Detailed settings for L1 and L2 servers, including the number of servers (value under `l1_servers`, `l2_servers`), storage media, capacity, and cache management strategies (e.g., `eviction_types` offering choices like FIFO, SIEVE, CLOCK, LRU) provide extensive control over server behavior and performance.

Origin server configurations: The `origin_config` section defines the characteristics of the origin server, such as its name and the simulated delay (delay set to 100 milliseconds) in retrieving content.

5.3.2 Searching Engine

This section delves into THEODON's second core module: the searching engine. It's designed to find near-optimal configurations balancing performance and cost in CDN environments.

Problem Formulation

Our focus is on a static CDN topology with a flexible configuration that encompasses variables such as the number of caches, cache size, and eviction algorithms, among others. Our objective is to leverage advanced optimization techniques to identify the most efficient configuration that meets our goals while adhering to specified constraints. Our goal is to optimize a list of M objective functions ($f^{(1)}(x)$, ..., $f^{(M)}(x)$) over a bounded search space $x \subset \chi$, where χ is the set of possible configuration spaces for a given CDN topology.

Solution with Multi-Objective Bayesian Optimization

Multi-Objective Bayesian Optimization (MOBO) [37, 111] is a framework to solve optimization problem when multiple conflicting objectives must be balanced. This method, unlike traditional single-objective approaches, provides a spectrum of Pareto-optimal solutions. In multi-objective optimization, our goal shifts from finding a single solution to identifying a Pareto set. This set allows for informed decisions, effectively balancing varying objectives. MOBO's implementation in THEODON is particularly adept at navigating the complex interplay between performance and cost.

Approaches within MOBO

In multi-objective bayesian optimization, methods can broadly be categoried into two main types based on their approach to handling multiple objectives: pareto-based methods and scalarization-based methods [92]. Pareto-based strategies focus on uncovering a range of non-dominated solutions, embodying trade-offs between objectives. Scalarization-based methods, conversely, simplify the multi-objective problem into a single-objective format using scalarization functions. We explored qEHVI and NParEGO, , representatives of these respective categories. Our evaluation demonstrates that qEHVI has higher effectiveness than NParEGO in the context of CDN.

Design Considerations and Decisions:

To leverage MOBO to find a good CDN configuration, we need to make several design decisions based on system constraints and requirements.

- **Starting points:** we randomly sample a few points (e.g., five) from the sample space using a quasi-random sequence.

- **Encoding configurations:** We encode all parameters from Table 5.1 into $\vec{x}$ to represent a CDN configuration. These parameters fall into two types: numerical and categorical. For categorical parameters, like eviction algorithms and load balancing strategies, we employ one-hot encoding. In this method, each category is represented by a binary vector. This approach enables the Gaussian Process (GP) to treat these inputs as continuous variables while still maintaining their distinct categorical characteristics.

- **Stopping conditions:** We define the stopping condition as follows: the search will stop once either the expected performance target is achieved and at least N (e.g. N=10) configurations have been observed. If neither performance target is met, the search process will be stopped after evaluating M (e.g. M=25) configurations.

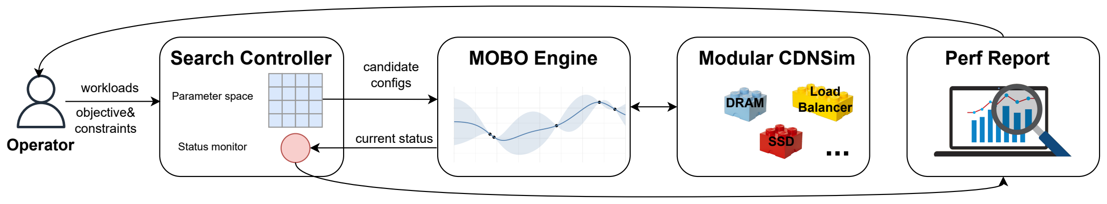


Figure 5.3: Architecture of THEODON's implementation.

5.4 Implementation

In this section, we discuss the implementation details of THEODON as shown in Fig. 5.3. It has four modules.

1. Controller: The controller orchestrates the entire CDN configuration selection process. To use THEODON, users supply a representative workload, the objective (e.g. minimizing byte miss ratio or cost), and the constraints (e.g. cost budget, preferred CDN topology parameter ranges, etc.). Based on these inputs, the controller obtains a list of candidate configurations and passes it to the MOBO engine. At the same time, controller triggers the modular simulator to construct the CDN that matches the user's requirements. Controller also monitor the current status and decides whether to finish the searching according to the stopping condition. The controller is implemented in Python.

2. Modular Simulator: The modular simulator in our framework constructs a CDN based on provided parameters and gathers performance statistics. For consistent and reliable results, THEODON adopts Lingua Franca [127], a reactor-oriented coordination language. This choice ensures deterministic concurrency, crucial for guaranteeing that simulation outcomes are reproducible, particularly in intricate and concurrent scenarios. We have implemented THEODON simulator in ≈17KLOC in Lingua Franca for core implementations and ≈1KLOC in YMAL for configuring parameters. Moreover, we integrate MQSim [194] for SSD simulation.

3. MOBO Engine: MOBO Engine is built on top of BoTorch [29] which is a modern programming framework for Bayesian optimization. We coordinate the

BoTorch framework with Theodon simulator in ≈1KLOC in Python.

4. Perf Analyzer: In Perf Analyzer, implemented in ≈3KLOC in Python, we collect and process performance statistics. This module is designed to analyze these stats and generate insightful plots and figures, aiding in the thorough examination and understanding of performance data.

5.5 Evaluation

In this study, our focus is on a single Point of Presence (PoP). We first study WikimediaCDN and Cloudflare, revealing that their byte miss ratios can be improved by 10% and 23%, respectively, compared to their default configurations. Then, we evaluate effectiveness of Theodon in identifying near-optimal configurations. This evaluation aims to achieve a balance between a low byte miss ratio and reduced costs.

5.5.1 Experimental Setup

Workloads. Our experiments use open-source traces from WikimediaCDN [208], comprising cache data for image and HTML pageview requests from two servers. These traces encompass 2,863 million requests involving 56 million objects. We replayed the traces in the simulator as a closed system with on-demand fill.

Objectives. We define two objectives for this study. The first objective is to minimize the byte miss ratio at a PoP, which reflects the traffic offloaded to the origin. The second objective is to minimize the hardware costs in a PoP. These costs are calculated based on the capacities of DRAM and SSDs used in the PoP. Recognizing the variability in actual pricing and the different internal prices offered by companies, we approximate the costs by assuming that the price of DRAM is 15 times that of SSD per GB.

Alternate algorithms. To demonstrate the efficacy of Theodon 's search engine, we evaluate it against the following algorithms:

- Random Search: This approach involves sampling configurations randomly within the parameter space. For this purpose, we utilized SOBOL, a quasi-random search algorithm implemented in BoTorch [29].
- Scalarization-based Search: We use qNParEGO, an efficient implementation based on random scalarization implemented in BoTorch.

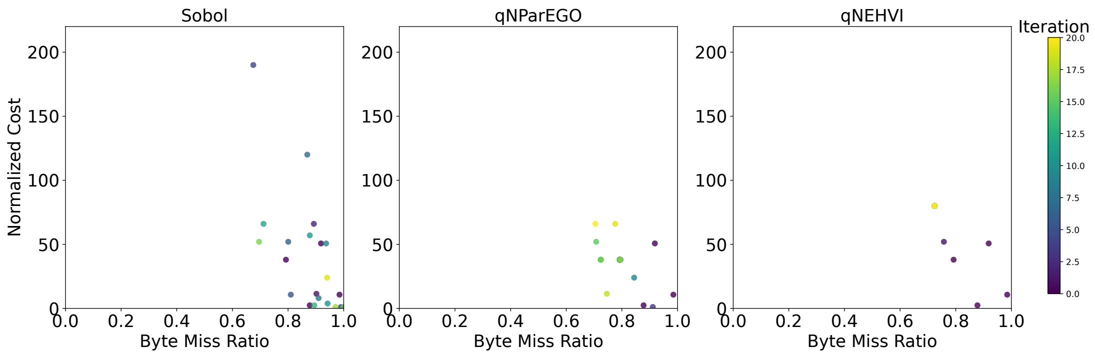


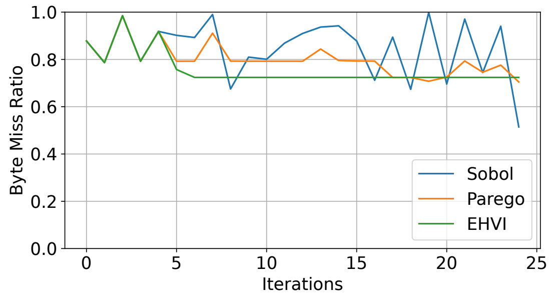
Figure 5.4: Comparing optimization process of random search (Sobol), scalarization-based search (qNParEGO), and pareto-based search (qNEHVI).

5.5.2 Effectiveness

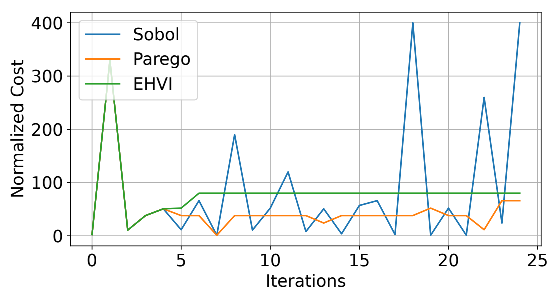
To assess the performance of THEODON's search engine, we tested Sobol, qNParEGO, and qNEHVI for the Cloudflare-like topology outlined in §5.2. Among the 33,792 configurations evaluated, a Pareto frontier was identified for byte miss ratio and hardware cost, as shown in Fig. 5.2. Our targets were set at a byte miss ratio of 0.8 and a hardware cost of 400. This search process is depicted in Fig. 5.4, where each point's color represents the Bayesian Optimization (BO) iteration at which it was identified.

The qNEHVI algorithm demonstrated a rapid ability to identify the Pareto frontier, with most of its evaluations clustering close to this frontier. This indicates its effectiveness in honing in on optimal solutions. On the other hand, qNParEGO, while also having many observations near the Pareto frontier, relies on optimizing random scalarizations. This approach is less structured in optimizing the Pareto front compared to aNEHVI, which explicitly and efficiently focuses on enhancing the frontier. In contrast, the Sobol algorithm, generating random points, resulted in fewer points in proximity to the Pareto front, reflecting its less targeted approach in this optimization task.

To gain deeper insights into the optimization process, we analyzed the two objectives—byte miss ratio and cost over iterations, as shown in Fig. 5.5. The results



(a) Byte miss ratio over iterations.



(b) Normalized cost over iterations.

Figure 5.5: Performance comparison of optimization algorithms over iterations.

show 25 iterations, with the first 5 iterations involving all three algorithms engaging in random search. Fig. 5.5a shows how the byte miss ratio evolves as the optimization progresses through 25 iterations. The Sobol algorithm exhibits fluctuations throughout the iterations, while both qNEHVI and qNParEGO show greater stability after the initial 10 iterations. Notably, qNEHVI maintains a consistently lower byte miss ratio compared to the other algorithms. In Fig. 5.5b, we observe a similar trend but with a notable difference in cost outcomes. qNParEGO achieves a lower cost than qNEHVI. This difference can be attributed to the more relaxed cost constraints we set in our optimization parameters.

5.5.3 Case Study

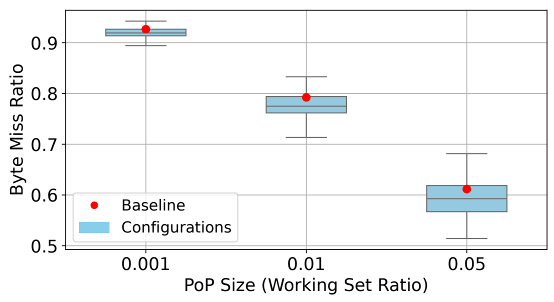
Next, we explore how Theodon can enhance real-world CDN performance by experimenting with different configurations. We initiate this exploration by simulating the WikimediaCDN and Cloudflare topologies, thoroughly examining an extensive array of configurations for each. Our analysis reveals promising opportunities for optimization in both CDN networks, suggesting that their performance could be significantly improved beyond the limitations of their existing default configurations. Following this, Theodon is employed to pinpoint configurations that optimally balance the trade-off between performance and cost.

WikimediaCDN. In WikimediaCDN's default configuration, the load balancing process operates at two levels. At the first level, traffic distribution is based on consistent hashing of the client IP. However, for the purpose of this discussion, we will assume the traffic is randomly distributed to a server. The second level of load balancing employs hashing based on the request URL. Regarding caching mechanisms, WikimediaCDN employs a two-tiered system. DRAM is utilized as the first level cache for its high-speed access, while SSDs serve as the second level cache, employing LRU and FIFO policies respectively. The specific configuration of a DRAM to SSD ratio of 0.2 is reflective of Wikimedia's public CDN hardware specifications [2].

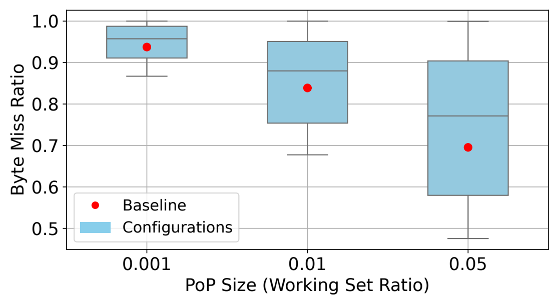
In our analysis of WikimediaCDN configurations, we embark on a detailed exploration of the parameter space. This entails systematically combining various settings across different parameters. Specially, our configuration parameters include:

- DRAM/SSD Ratios: We consider eight distinct ratios, ranging from 0.001 to 0.5. The options tested were 0.001, 0.005, 0.01, 0.1, 0.2, 0.3, 0.4, and 0.5.
- Number of L1 Caches: There are eleven possible configurations for L1 caches, varying from 1 to 10 and including an extended option of 30.
- Number of L2 Caches: This parameter mirror the L1 cache options, with eleven configurations also ranging from 1 to 30.
- L1 Eviction Policies: Four eviction strategies we evaluate for L1 caches, including CLOCK, FIFO, LRU, and SIEVE.
- L2 Eviction Policies: Identical to the L1 options.
- L1 load balancing: We explore two methods, random distribution and URL-based hashing ('random', 'url_hash').

Additionally, we tailor the PoP capacity in relation to the working set (total bytes) of the input trace. The PoP capacity is set as a fraction of the working set, specifically



(a) WikimediaCDN



(b) Cloudflare

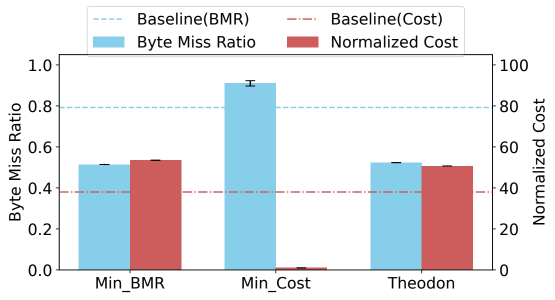
Figure 5.6: The box shows the byte miss ratio across 92,928 configurations for Wikimedia and 33,792 configurations for Cloudflare.

at 0.001, 0.01, and 0.05, to understand how varying levels of resource allocation affect the system's functionality.

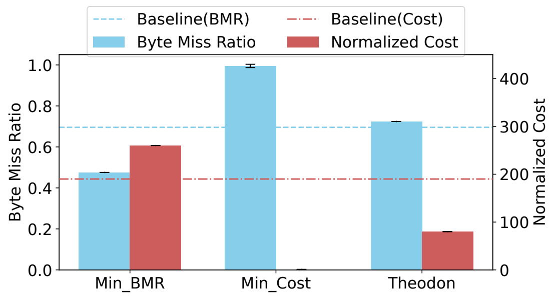
The total number of simulated scenarios is determined by the product of the options across each category, leading to 92,928 unique configurations. As shown in Fig. 5.6a, we can observe up to a 10% improvement in terms of byte miss ratio for the same PoP capacity.

Cloudflare. Then, we conduct the similar experiments for Cloudflare topology. In alignment with WikimediaCDN's approach, Cloudflare also employs a two-level load balancing strategy. For the first level, we implement random distribution, while the second level utilizes hashing based on the request URL. Despite similarities in using a two-tier caching system comprising DRAM and SSD, Cloudflare adopts a distinct caching strategy. Instead of caching popular objects in DRAM, Cloudflare stores these objects in SSD. The system tracks the number of hits an object receives in DRAM, promoting it to SSD after a single hit. This default promotion threshold is set at one. Additionally, Cloudflare employs the LRU eviction policy for both DRAM and SSD caches. Since we lack specific information about the ratio between DRAM and SSD used in Cloudflare, we adopt the same ratio as in WikimediaCDN's setting, which is 0.2.

In our analysis of Cloudflare configurations, we use a set of parameters that are



(a) WikimediaCDN



(b) Cloudflare

Figure 5.7: Byte miss ratio and cost comparison for WikimediaCDN and Cloudflare with different optimizing schemes. "Min_BMR" and "Min_Cost" focus on optimizing either byte miss ratio or cost, respectively, while THEODON optimizes both. The dotted lines are the results of default setting for WikimediaCDN and Cloudflare.

largely consistent with those used for Wikimedia configurations. However, there are a few key differences in the parameters specific to Cloudflare, which are as follows:

- Number of L1 and L2 Caches: This parameter has the same range varying from 1 to 10 and including an extended option of 30, but the number of L1 is maintained same as the number of L2 because of the promotion strategy.
- Number of hits for promotion: We consider four distinct values, ranging from 1 to 4.

Consequently, the total number of simulated experiments is therefore 33,792. As shown in Fig. 5.6b, we can observe up to a 23% improvement in terms of byte miss ratio.

Trade-off between performance and cost.

To explore the trade-off between performance and cost in CDN configurations, we use three optimization schemes. Min_BMR focuses on minimizing byte miss ratio, Min_Cost targets cost reduction, and THEODON seeks a balanced outcome between the two.

Fig. 5.7 presents the results for both WikimediaCDN and Cloudflare, illustrating the capabilities of each scheme. Remarkably, THEODON achieves a balance between the objectives in just 25 iterations of simulations. For WikimediaCDN, both Min_BMR and THEODON manage to substantially lower the byte miss ratio – by up to 30% – albeit at a cost increase of approximately 1.3× the default setting. In contrast, for Cloudflare, THEODON displays a more pronounced improvement in cost efficiency, given our constraint of maintaining costs below 200. While THEODON maintains a byte miss ratio comparable to the default setting, it also achieves a significant cost reduction, lowering expenses by about 2.4×.

5.6 Discussion

Network layer simulation. We recognize a notable limitation in our current simulation framework: the omission of the network layer, a crucial component of any CDN. The influence of the network layer on latency and data delivery is profound, with significant implications for the efficacy of caching strategies and the overall user experience. To address this gap, we plan to incorporate Mahimahi, a framework capable of accurately simulating network conditions, including variables such as bandwidth, latency, and packet loss [145]. The integration of Mahimahi will enable us to evaluate our caching strategies in an environment that closely replicates the complexities and variabilities of real-world internet traffic, providing a more robust assessment of performance.

Cost model. The cost metric used in our current evaluation is based solely on the cost of DRAM and SSDs within a PoP, offering a somewhat limited perspective on the comprehensive cost structure of a CDN. To develop a more inclusive cost model, we intend to include additional factors that significantly influence the total operating costs of a CDN. These factors encompass the electricity costs, which vary widely based on geographical location and scale, as well as the costs related to bandwidth, which are subject to fluctuations based on data transfer volumes and network agreements. Furthermore, we aim to consider the expenses tied to maintenance and the amortization of infrastructure over time.

5.7 Related Work

System parameter tuning. Many prior works have explored the configuration tuning for large-scale complex systems. There are two categories of optimizing approaches those work commonly use: Multi-Armed Bandit approach and Bayesian Optimization. For the first category, Dremel [236] adaptively and quickly configures RocksDB with strategies of feature fusion and online tuning with multi-armed bandit models to achieves igh performance while adapting to specific workload and hardware conditions. Configanator [144] aims to learn the optimal configuration parameters for web server configurations to improve CDN performance. Its learning algorithm consists of a contextual multi-armed bandit with three arms, gaussian process, epsilon-bandit, and decision tree, to avoid sub-optimal tuning. For the second category, CherryPick [20] leverages Bayesian Optimization to unearth the best cloud configurations for big data analytics. CLITE [157] also uses Bayesian Optimization based multi-resource partitioning technique to satisfy QoS requirements and maximize the performances.

CDN simulators. Researchers commonly employ simulation tools to assess the performance of a CDN, and some even conduct experiments on real platforms like PlanetLab [54]. There is an array of network simulators [10] available that can be utilized to simulate a CDN's performance. Additionally, specialized CDN simulation systems [109, 188] offer a more realistic approach, beneficial to both the research community and CDN developers, for evaluating CDN performance and testing various CDN policies. However, these simulators tend to focus more on network conditions and less on the fundamental configuration space at the architectural level. They also lack the flexibility required to investigate the trade-offs between performance and cost, which is a critical aspect in the optimization of CDNs.

5.8 Conclusion

We present THEODON, a framework that models CDN architectures through modular simulations. Leveraging empirical CDN workloads,THEODON enables us to find configurations that strike a balance between performance and cost. We evaluate THEODON in two real-world cases: WikimediaCDN and Cloudflare with public traces from WikimediaCDN. Our findings indicate that THEODON effectively identifies configurations that can improve performance by up to 10% and 23% compared to the default settings of WikimediaCDN and Cloudflare, respectively. A significant highlight is THEODON's ability to maintain a byte miss ratio akin to Cloudflare's default while efficiently identifying configurations that considerably cut costs, reducing expenses by about 2.4×.

Chapter 6

Conclusion

This dissertation demonstrates the approach to data-driven performance modeling in complex networked systems, anchored in two main principles. The first principle emphasizes the decomposition of the entire system into key components with clearly interpretable interactions. The second principle involves utilizing empirical system data to shape the modeling process and inform decision-making. Applying these two key principles,we have carried out data-driven performance modeling across three diverse and broadly complicated systems: microservice-based applications, large-scale web cache systems and CDNs, which we summarize below.

LatenSeer is a data-driven modeling framework for estimating end-to- end latency distributions in microservice-based web applications. It captures the complex relationships between services, representing these in a novel abstraction — invocation graph. This tool enables developers to explore various what-if scenarios, facilitating debugging and performance enhancement in their applications. LatenSeer predicts latency within a 5.35% error, outperforming the state-of-the-art that has an error rate of more than 9.5%.

Sieve is a cache primitive that is simpler than LRU and provides better than state-of-the-art efficiency and scalability for web cache workloads — a critical component of CDN architecture. The development of Sieve is closely informed by real-world production web cache workloads, utilizing data on access patterns, such as the frequency of cache item visits, to make strategic decisions. Implemented in five production cache libraries, Sieve is notable for its ease of integration, requiring less than 20 lines of code modification on average. Our evaluation on 1559 cache traces from 7 sources shows that Sieve achieves up to 63.2% lower miss ratio than ARC. Moreover, Sieve has a lower miss ratio than 9 state-of-the-art algorithms on more than 45% of the 1559 traces, while the next best algorithm only has a lower miss ratio on 15%.

Theodon is a framework that models CDN architectures through modular simulations, effectively dissecting a CDN's topology into interconnected components that each represent a fundamental element of the CDN infrastructure. Leveraging real-world CDN workloads, Theodon identifies configurations that adeptly balance performance with cost. In practical applications, by simulating two real-world CDN systems —WikimediaCDN and Cloudflare — Theodon has revealed substantial improvements, achieving reductions in byte miss ratio of up to 10% and 23%, respectively. Moreover, Theodon has proven its ability to significantly cut costs, reducing expenses by about 2.4× in the case of Cloudflare, while still matching the performance levels of the default settings.

The contributions of this dissertation — data-driven performance modeling in complex networked systems — is vital in understanding and enhancing system reliability and efficiency. This dissertation showcases the application of two principles across three distinct complex systems microservice-based applications, large-scale web cache systems, and CDNs. With the emergence of new complex networked systems like serverless computing, the insights derived from the design of LatenSeer, Sieve, and Theodon are valuable in enabling performance analysis in these evolving systems.

6.1 Future Directions

Our work also opens several interesting avenues for future research. This thesis has discussed the data-driven performance modeling in three networked systems, resulting tools and algorithm. However, there are many opportunities in each work to explore to further facilitate improving the system reliability and efficiency.

Data-driven scheduling For microservice-based applications, our current work has focused on estimating the end-to-end latency and identifying latency slack of microservices. LatenSeer achieves these by capturing the causal dependencies between the services. With these abstraction, LatenSeer can further aim to reduce communication costs associated with expensive Remote Procedure Calls (RPCs). Currently, service instances are often placed on machines somewhat randomly, which can lead to inefficiencies in communication, especially among services frequently involved in RPC calls. LatenSeer could be expanded to pinpoint service pairs that frequently engage in RPC calls and are thus prime candidates for co-location either on the same machine or within the same rack. Such strategic placement has the potential to lower communication overhead and improve end-to-end latency. There are many open questions to explore, including strategies to minimize communication overhead within the same machine and how to schedule RPC calls in response to fluctuating traffic.

Heterogeneity-aware considerations As datacenter hardware becomes more heterogeneous, there are many opportunities arise to leverage this diversity more effectively. THEODON currently offers a limited parameter space for hardware configurations, primarily focusing on basic distinctions between DRAM and SSD. To optimize utilization further, we could expand this parameter space to include detailed listings of specific hardware types. Rather than adhering to the traditional method of traffic distribution across servers, a more nuanced approach could be adopted. This approach would consider the status of the hardware, adjusting traffic distribution based on traffic patterns to make the most of the deployed hardware's capabilities. Another direction worth exploring is the clustering of hardware into high-end and low-end categories, each with different performance levels and resource consumption profiles. Traffic could be dynamically distributed across these clusters of machines, aiming to fulfill user requirements more efficiently while simultaneously reducing costs.

Workload benchmark. SIEVE has been evaluated through an extensive study involving various workloads and has demonstrated its efficiency in web workloads, particularly those following a Zipfian distribution. However, comprehensively understanding the characteristics of all cache workloads is crucial to facilitate the design of future eviction algorithms that can adequately meet evolving demands. The diverse workloads encountered by distributed caching systems today pose a distinctive challenge, necessitating careful analysis and classification due to their variety. Building upon SIEVE's initial research on web cache workloads, it's important to expand this scope to include both storage workloads and CPU cache workloads. By incorporating a broader set of representative traces, we can enhance our evaluation of future cache eviction designs, ensuring they are well-tuned to the dynamic needs of contemporary computing environments.

Bibliography

[1] Alibaba block-trace. https://github.com/alibaba/block-traces. Accessed: 2023-01-12.

[2] Cdn/hardware. https://wikitech.wikimedia.org/wiki/CDN/Hardware. Accessed: 2024-01-17.

[3] lru-dict. https://github.com/amitdev/lru-dict. Accessed: 2023-04-27.

[4] lru-rs. https://github.com/jeromefroe/lru-rs. Accessed: 2023-04-27.

[5] Memcached - a distributed memory object caching system. http://memcached.org/. Accessed: 2023-04-27.

[6] mnemonist. https://github.com/Yomguithereal/mnemonist. Accessed: 2023-04-27.

[7] pelikan. https://github.com/pelikan-io/pelikan. Accessed: 2023-04-27.

[8] Redis. http://redis.io/. Accessed: 2023-02-06.

[9] Running cachebench with the trace workload. https://cachelib.org/docs/Cache_Library_User_Guides/Cachebench_FB_HW_eval. Accessed: 2023-02-12.

[10] Network simulators and related links. https://ee.lbl.gov/kfall/netsims.html, 2007. Accessed: 2024-03-13.

[11] Analytics/data lake/traffic/caching. https://wikitech.wikimedia.org/wiki/Analytics/Data_Lake/Traffic/Caching, 2023. Accessed: 2024-01-17.

[12] golang-fifo. https://github.com/scalalang2/golang-fifo, 2023. Accessed: 2024-03-13.

[13] Reduce latency and increase cache hits with regional tiered cache. https://blog.cloudflare.com/introducing-regional-tiered-cache, 2023. Accessed: 2024-01-17.

[14] Ristretto. https://github.com/dgraph-io/ristretto, 2023. Accessed: 2024-03-13.

[15] The cloudflare global network. https://www.cloudflare.com/network/, 2024. Accessed: 2024-03-13.

[16] Dragonfly. https://github.com/dragonflydb/dragonfly/blob/main/src/core/compact_object.h#L124, 2024. Accessed: 2024-03-13.

[17] Skiftos. https://github.com/skift-org/skift/blob/main/src/libs/karm-base/sieve.h, 2024. Accessed: 2024-03-13.

[18] Vijay Kumar Adhikari, Yang Guo, Fang Hao, Matteo Varvello, Volker Hilt, Moritz Steiner, and Zhi-Li Zhang. Unreeling netflix: Understanding and improving multi-CDN movie delivery. In *2012 Proceedings IEEE INFOCOM*, pages 1620–1628, Orlando, FL, USA, March 2012. IEEE.

[19] Akamai Technologies. Akamai Online Retail Performance Report: Milliseconds Are Critical. Retrieved April 2022 from https:

`//www.akamai.com/newsroom/press-release/akamai-releases-spring-2017-state-of-online-retail-performance-report`, April 2017.

[20] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. CherryPick: Adaptively unearthing the best cloud configurations for big data analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI'17)*, pages 469–482, USA, 2017. USENIX Association.

[21] Mohammad Alrifai and Thomas Risse. Combining global optimization with local selection for efficient qos-aware service composition. In *Proceedings of the 18th international conference on World wide web*, pages 881–890, 2009.

[22] Mohammad Alrifai, Dimitrios Skoutas, and Thomas Risse. Selecting skyline services for qos-based web service composition. In *Proceedings of the 19th international conference on World wide web*, pages 11–20, 2010.

[23] Apache. Apache traffic server. `https://trafficserver.apache.org/`. Accessed: 2023-02-06.

[24] Danilo Ardagna and Barbara Pernici. Adaptive service composition in flexible processes. *IEEE Transactions on software engineering*, 33(6):369–384, 2007.

[25] Ismail Ari, Ahmed Amer, Robert B Gramacy, Ethan L Miller, Scott A Brandt, and Darrell DE Long. ACME: Adaptive Caching Using Multiple Experts. In *WDAS*, volume 2, pages 143–158, 2002.

[26] Emre Ates, Lily Sturmann, Mert Toslali, Orran Krieger, Richard Megginson, Ayse K Coskun, and Raja R Sambasivan. An automated, cross-layer instrumentation framework for diagnosing performance problems in distributed applications.

In *Proceedings of the ACM Symposium on Cloud Computing (SOCC'19)*, pages 165–170, New York, NY, USA, 2019. Association for Computing Machinery.

[27] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-Scale Key-Value Store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 53–64, New York, NY, USA, 2012. Association for Computing Machinery.

[28] Nirav Atre, Justine Sherry, Weina Wang, and Daniel S. Berger. Caching with Delayed Hits. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, pages 495–513, New York, NY, USA, 2020. Association for Computing Machinery.

[29] Maximilian Balandat, Brian Karrer, Daniel Jiang, Samuel Daulton, Ben Letham, Andrew G Wilson, and Eytan Bakshy. Botorch: A framework for efficient monte-carlo bayesian optimization. *Advances in neural information processing systems*, 33:21524–21538, 2020.

[30] Sorav Bansal and Dharmendra S. Modha. CAR: Clock with Adaptive Replacement. In *3rd USENIX Conference on File and Storage Technologies*, FAST'04, 2004.

[31] Paul Barford and Mark Crovella. Critical path analysis of tcp transactions. *ACM SIGCOMM Computer Communication Review*, 30(4):127–138, 2000.

[32] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using Magpie for request extraction and workload modelling. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI'04)*, volume 4, pages 18–33, New York, NY, USA, 2004. Association for Computing Machinery.

[33] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. LHD: Improving cache hit rate by maximizing hit density. In *15th USENIX symposium on networked systems design and implementation*, NSDI'18, pages 389–403, 2018.

[34] Nathan Beckmann and Daniel Sanchez. Talus: A simple way to remove cliffs in cache performance. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture*, HPCA'15, pages 64–75, Burlingame, CA, USA, February 2015. IEEE.

[35] Nathan Beckmann and Daniel Sanchez. Maximizing Cache Performance Under Uncertainty. In *2017 IEEE International Symposium on High Performance Computer Architecture*, HPCA'17, pages 109–120, Austin, TX, February 2017. IEEE.

[36] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[37] Syrine Belakaria, Aryan Deshwal, and Janardhan Rao Doppa. Max-value entropy search for multi-objective bayesian optimization. *Advances in neural information processing systems*, 32, 2019.

[38] Benjamin Berg, Daniel S. Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, and Gregory R. Ganger. The CacheLib caching engine: Design and experiences at scale. In *14th USENIX symposium on operating systems design and implementation*, OSDI'20, pages 753–768. USENIX Association, November 2020.

[39] Daniel S. Berger, Benjamin Berg, Timothy Zhu, Siddhartha Sen, and Mor Harchol-Balter. RobinHood: Tail latency aware caching – dynamic reallocation from Cache-Rich to Cache-Poor. In *13th USENIX symposium on operating*

systems design and implementation, OSDI'18, pages 195–212, Carlsbad, CA, October 2018. USENIX Association.

[40] Daniel S Berger, Ramesh K Sitaraman, and Mor Harchol-Balter. AdaptSize: Orchestrating the hot object memory cache in a content delivery network. In *14th USENIX symposium on networked systems design and implementation*, NSDI'17, pages 483–498, 2017.

[41] Adit Bhardwaj and Vaishnav Janardhan. Pecc: Prediction-error correcting cache. In *Workshop on ML for Systems at NeurIPS*, volume 32, 2018.

[42] Aaron Blankstein, Siddhartha Sen, and Michael J. Freedman. Hyperbolic caching: Flexible caching for web applications. In *2017 USENIX annual technical conference*, ATC'17, pages 499–511, Santa Clara, CA, July 2017. USENIX Association.

[43] David Böhme, Markus Geimer, Lukas Arnold, Felix Voigtlaender, and Felix Wolf. Identifying the root causes of wait states in large-scale parallel applications. *ACM Transactions on Parallel Computing (TOPC)*, 3(2):1–24, 2016.

[44] Bradfitz. group cache. `https://github.com/golang/groupcache`. Accessed: 2023-02-06.

[45] L. Breslau, Pei Cao, Li Fan, G. Phillips, and S. Shenker. Web caching and Zipf-like distributions: evidence and implications. In *Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies*, pages 126–134 vol.1, New York, NY, USA, 1999. IEEE.

[46] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. On the implications of Zipf's law for web caching. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 1998.

[47] Daniel Byrne, Nilufer Onder, and Zhenlin Wang. mPart: miss-ratio curve guided partitioning in key-value stores. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on Memory Management*, ISMM'18, pages 84–95, Philadelphia PA USA, June 2018. ACM.

[48] Pei Cao and Sandy Irani. Cost-Aware WWW Proxy Caching Algorithms. In *USENIX Symposium on Internet Technologies and Systems*, USITS'97, Monterey, CA, December 1997. USENIX Association.

[49] Laura Carnevali, Riccardo Reali, and Enrico Vicario. Compositional evaluation of stochastic workflows for response time analysis of composite web services. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, pages 177–188, 2021.

[50] Richard W. Carr and John L. Hennessy. WSCLOCK: a simple and effective algorithm for virtual memory management. In *Proceedings of the eighth ACM symposium on Operating systems principles*, SOSP '81, pages 87–95, New York, NY, USA, December 1981. Association for Computing Machinery.

[51] H. Che, Z. Wang, and Y. Tung. Analysis and design of hierarchical Web caching systems. In *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No.01CH37213)*, volume 3, pages 1416–1424, Anchorage, AK, USA, 2001. IEEE.

[52] Mike Y Chen, Anthony Accardi, Emre Kiciman, Jim Lloyd, Dave Patterson, Armando Fox, and Eric Brewer. Path-based failure and evolution management. In *1st USENIX Symposium on Networked Systems Design & Implementation (NSDI'04)*, pages 23–23, USA, 2004. USENIX Association.

[53] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F Wenisch.

The Mystery Machine: End-to-end Performance Analysis of Large-scale Internet Services. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*, page 217–231, USA, 2014. USENIX Association.

[54] Brent Chun, David Culler, Timothy Roscoe, Andy Bavier, Larry Peterson, Mike Wawrzoniak, and Mic Bowman. Planetlab: an overlay testbed for broad-coverage services. *ACM SIGCOMM Computer Communication Review*, 33(3):3–12, 2003.

[55] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. Cliffhanger: Scaling performance cliffs in web memory caches. In *13th USENIX symposium on networked systems design and implementation*, NSDI'16, pages 379–392, 2016.

[56] Asaf Cidon, Daniel Rushton, Stephen M. Rumble, and Ryan Stutsman. Memshare: a dynamic multi-tenant key-value cache. In *2017 USENIX annual technical conference*, ATC'17, pages 321–334, Santa Clara, CA, July 2017. USENIX Association.

[57] Jeremy Cloud. Decomposing twitter: Adventures in service-oriented architecture. `https://www.infoq.com/presentations/twitter-soa/`, 2013. Accessed: 2023-05-21.

[58] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[59] Fernando J Corbato. A paging experiment with the multics system. Technical report, MASSACHUSETTS INST OF TECH CAMBRIDGE PROJECT MAC, 1968.

[60] Samuel Daulton, Maximilian Balandat, and Eytan Bakshy. Parallel bayesian op-

timization of multiple noisy objectives with expected hypervolume improvement. *Advances in Neural Information Processing Systems*, 34:2187–2200, 2021.

[61] Vadim Denisov, Dirk Fahland, and Wil MP van der Aalst. Predictive performance monitoring of material handling systems using the performance spectrum. In *2019 International Conference on Process Mining (ICPM)*, pages 137–144. IEEE, 2019.

[62] Peter J Denning. The working set model for program behavior. *Communications of the ACM*, 11(5):323–333, 1968.

[63] Meta developers. Cachelib - pluggable caching engine to build and scale high performance cache services. `https://cachelib.org/`. Accessed: 2023-04-06.

[64] Meta developers. Distributed mutex. `https://github.com/facebook/folly/blob/2c00d14adb9b632936f3abfbf741373871cd64a6/folly/synchronization/DistributedMutex.h`. Accessed: 2023-04-27.

[65] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, S.H. Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. LRFU: a spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE Transactions on Computers*, 50(12):1352–1361, December 2001.

[66] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019.

[67] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, et al. The design and operation of CloudLab. In *2019 USENIX Annual Technical Conference (USENIX ATC'19)*, pages 1–14, New York, NY, USA, 2019. Association for Computing Machinery.

[68] Brian Eaton, Jeff Stewart, Jon Tedesco, and N Cihan Tas. Distributed latency profiling through critical path tracing: Cpt can provide actionable and precise latency analysis. *Queue*, 20(1):40–79, 2022.

[69] Gil Einziger, Ohad Eytan, Roy Friedman, and Ben Manes. Adaptive Software Cache Management. In *Proceedings of the 19th International Middleware Conference*, pages 94–106, Rennes France, November 2018. ACM.

[70] Gil Einziger, Ohad Eytan, Roy Friedman, and Benjamin Manes. Lightweight robust size aware cache management. *ACM Transactions on Storage*, 18(3), August 2022.

[71] Gil Einziger, Roy Friedman, and Ben Manes. TinyLFU: A Highly Efficient Cache Admission Policy. *ACM Transactions on Storage*, 13(4):1–31, December 2017.

[72] Assaf Eisenman, Asaf Cidon, Evgenya Pergament, Or Haimovich, Ryan Stutsman, Mohammad Alizadeh, and Sachin Katti. Flashield: a hybrid key-value cache that controls flash write amplification. In *16th USENIX symposium on networked systems design and implementation*, NSDI'19, pages 65–78, Boston, MA, February 2019. USENIX Association.

[73] Assaf Eisenman, Maxim Naumov, Darryl Gardner, Misha Smelyanskiy, Sergey Pupyrev, Kim Hazelwood, Asaf Cidon, and Sachin Katti. Bandana: Using non-volatile memory for storing deep learning models. In A. Talwalkar, V. Smith,

and M. Zaharia, editors, *Proceedings of machine learning and systems*, volume 1 of *mlsys'20*, pages 40–52, 2019.

[74] Simon Eismann, Long Bui, Johannes Grohmann, Cristina Abad, Nikolas Herbst, and Samuel Kounev. Sizeless: Predicting the optimal size of serverless functions. In *Proceedings of the 22nd International Middleware Conference*, pages 248–259, 2021.

[75] Simon Eismann, Johannes Grohmann, Erwin Van Eyk, Nikolas Herbst, and Samuel Kounev. Predicting the costs of serverless workflows. In *Proceedings of the ACM/SPEC international conference on performance engineering*, pages 265–276, 2020.

[76] Joyce El Hadad, Maude Manouvrier, and Marta Rukoz. Tqos: Transactional and qos-aware selection algorithm for automatic web service composition. *IEEE Transactions on Services Computing*, 3(1):73–85, 2010.

[77] Úlfar Erlingsson, Marcus Peinado, Simon Peter, Mihai Budiu, and Gloria Mainar-Ruiz. Fay: Extensible distributed tracing from kernels to clusters. *ACM Transactions on Computer Systems (TOCS)*, 30(4):1–35, 2012.

[78] Ohad Eytan, Danny Harnik, Effi Ofer, Roy Friedman, and Ronen Kat. It's time to revisit LRU vs. FIFO. In *12th USENIX workshop on hot topics in storage and file systems*, hotStorage'20. USENIX Association, July 2020.

[79] Bin Fan, David G Andersen, and Michael Kaminsky. MemC3: Compact and concurrent MemCache with dumber caching and smarter hashing. In *10th USENIX symposium on networked systems design and implementation*, NSDI'13, pages 371–384, 2013.

[80] Brian Fields, Shai Rubin, and Rastislav Bodik. Focusing processor policies via critical-path prediction. In *Proceedings of the 28th annual international*

symposium on Computer architecture, pages 74–85, New York, NY, USA, 2001. Association for Computing Machinery.

[81] Tobias Flach, Nandita Dukkipati, Andreas Terzis, Barath Raghavan, Neal Cardwell, Yuchung Cheng, Ankur Jain, Shuai Hao, Ethan Katz-Bassett, and Ramesh Govindan. Reducing web latency: The virtue of gentle aggression. In *Proceedings of the 2013 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM'13)*, SIGCOMM '13, pages 159–170, New York, NY, USA, 2013. ACM.

[82] Philippe Flajolet, Daniele Gardy, and Loÿs Thimonier. Birthday paradox, coupon collectors, caching algorithms and self-organizing search. *Discrete Applied Mathematics*, 39(3):207–229, 1992.

[83] Rodrigo Fonseca, George Porter, Randy H Katz, and Scott Shenker. X-trace: A pervasive network tracing framework. In *4th USENIX Symposium on Networked Systems Design & Implementation (NSDI'07)*, 2007.

[84] Nicole Forsgren, Dustin Smith, Jez Humble, and Jessie Frazelle. 2019 accelerate state of devops report. Technical report, Google.Inc, 2019.

[85] Thomas R Frieden. Evidence for health decision making—beyond randomized, controlled trials. *New England Journal of Medicine*, 377(5):465–475, 2017.

[86] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*, pages 3–18, New York, NY, USA, 2019. Association for Computing Machinery.

[87] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*, pages 19–33, New York, NY, USA, 2019. Association for Computing Machinery.

[88] Fănică Gavril. Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph. *SIAM Journal on Computing*, 1(2):180–187, 1972.

[89] Phillipa Gill, Martin Arlitt, Zongpeng Li, and Anirban Mahanti. Youtube traffic characterization: a view from the edge. In *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, pages 15–28, 2007.

[90] Adam Gluck. Introducing domain-oriented microservice architecture. `https://www.uber.com/blog/microservice-architecture/`, 2020. Accessed: 2023-05-21.

[91] Xiaoming Gu and Chen Ding. On the theory and potential of lru-mru collaborative cache management. *SIGPLAN Not.*, 46(11):43–54, jun 2011.

[92] Nyoman Gunantara. A review of multi-objective optimization: Methods and its applications. *Cogent Engineering*, 5(1):1502242, 2018.

[93] Lei Guo, Enhua Tan, Songqing Chen, Zhen Xiao, and Xiaodong Zhang. The stretched exponential distribution of internet media access patterns. In *Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing*, pages 283–294, 2008.

[94] Syed Hasan, Sergey Gorinsky, Constantine Dovrolis, and Ramesh K Sitaraman.

Trade-offs in optimizing the cache deployments of cdns. In *IEEE INFOCOM 2014-IEEE conference on computer communications*, pages 460–468. IEEE, 2014.

[95] Herodotos Herodotou and Shivnath Babu. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. *Proceedings of the VLDB Endowment*, 4(11):1111–1122, 2011.

[96] Xiameng Hu, Xiaolin Wang, Lan Zhou, Yingwei Luo, Chen Ding, and Zhenlin Wang. Kinetic modeling of data eviction in cache. In *2016 USENIX annual technical conference*, ATC'16, pages 351–364, Denver, CO, June 2016. USENIX Association.

[97] Xinyue Hu, Eman Ramadan, Wei Ye, Feng Tian, and Zhi-Li Zhang. Raven: belady-guided, predictive (deep) learning for in-memory and content caching. In *Proceedings of the 18th International Conference on emerging Networking EXperiments and Technologies*, CoNEXT '22, pages 72–90, New York, NY, USA, November 2022. Association for Computing Machinery.

[98] Lexiang Huang and Timothy Zhu. tprof: Performance profiling via structural aggregation and automated analysis of distributed systems traces. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 76–91, New York, NY, USA, 2021. Association for Computing Machinery.

[99] Qi Huang, Ken Birman, Robbert van Renesse, Wyatt Lloyd, Sanjeev Kumar, and Harry C. Li. An analysis of Facebook photo caching. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 167–181, New York, NY, USA, November 2013. Association for Computing Machinery.

[100] Stratis Ioannidis, Laurent Massoulie, and Augustin Chaintreau. Distributed caching over heterogeneous mobile networks. In *Proceedings of the ACM SIG-*

METRICS international conference on Measurement and modeling of computer systems, SIGMETRICS'10, pages 311–322, 2010.

[101] Stratis Ioannidis and Edmund Yeh. Adaptive Caching Networks with Optimality Guarantees. In *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science*, SIGMETRICS'16, pages 113–124, Antibes Juan-les-Pins France, June 2016. ACM.

[102] Jaeger: Open Source, End-to-End Distributed Tracing. `https://www.jaegertracing.io/`, 2023. Accessed: 2023-05-21.

[103] Song Jiang, Feng Chen, and Xiaodong Zhang. CLOCK-Pro: an effective improvement of the CLOCK replacement. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATC'05, page 35, USA, April 2005. USENIX Association.

[104] Song Jiang and Xiaodong Zhang. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *ACM SIGMETRICS Performance Evaluation Review*, volume 30 of *SIGMETRICS'02*, pages 31–42, June 2002.

[105] Yurong Jiang, Lenin Ravindranath Sivalingam, Suman Nath, and Ramesh Govindan. WebPerf: Evaluating what-if scenarios for cloud-hosted web applications. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 258–271, New York, NY, USA, 2016. Association for Computing Machinery.

[106] Theodore Johnson and Dennis Shasha. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB'94, pages 439–450, San Francisco, CA, USA, September 1994. Morgan Kaufmann Publishers Inc.

[107] Diviyan Kalainathan, Olivier Goudet, Isabelle Guyon, David Lopez-Paz, and MichÃ¨le Sebag. Structural agnostic modeling: Adversarial learning of causal graphs. *Journal of Machine Learning Research*, 23(219):1–62, 2022.

[108] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, et al. Canopy: An end-to-end performance tracing and analysis system. In *Proceedings of the 26th symposium on operating systems principles*, pages 34–50, New York, NY, USA, 2017. Association for Computing Machinery.

[109] Jussi Kangasharju, James Roberts, and Keith W Ross. Object replication strategies in content distribution networks. *Computer Communications*, 25(4):376–383, 2002.

[110] R. Karedla, J.S. Love, and B.G. Wherry. Caching strategies to improve disk system performance. *Computer*, 27(3):38–46, March 1994.

[111] Nazan Khan, David E Goldberg, and Martin Pelikan. Multi-objective bayesian optimization algorithm. In *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation*, pages 684–684, 2002.

[112] Kyle Kingsbury. The trouble with timestamps. `https://aphyr.com/posts/299-the-trouble-with-timestamps`, 2013. Accessed: 2023-05-30.

[113] Ricardo Koller and Raju Rangaswami. I/o deduplication: Utilizing content similarity to improve i/o performance. *ACM Transactions on Storage (TOS)*, 6(3):1–26, 2010.

[114] Darja Krushevskaja and Mark Sandler. Understanding latency variations of black box services. In *Proceedings of the 22nd International Conference on*

the World Wide Web (WWW'13), pages 703–714, New York, NY, USA, 2013. Association for Computing Machinery.

[115] Pedro Las-Casas, Giorgi Papakerashvili, Vaastav Anand, and Jonathan Mace. Sifter: Scalable sampling for distributed traces, without feature engineering. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC'19)*, pages 312–324, 2019.

[116] Pedro Las-Casas, Giorgi Papakerashvili, Vaastav Anand, and Jonathan Mace. Sifter: Scalable sampling for distributed traces, without feature engineering. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, page 312–324, New York, NY, USA, 2019. Association for Computing Machinery.

[117] Stephen Lavenberg. *Computer performance modeling handbook*. Elsevier, 1983.

[118] Cate Lawrence. Deployment frequency – a key metric in devops. `https://humanitec.com/blog/deployment-frequency-key-metric-in-devops`, 2021. Accessed: 2023-05-30.

[119] Cong Li. DLIRS: Improving Low Inter-Reference Recency Set Cache Replacement Policy with Dynamics. In *Proceedings of the 11th ACM International Systems and Storage Conference*, SYSTOR '18, pages 59–64, New York, NY, USA, June 2018. Association for Computing Machinery.

[120] Conglong Li and Alan L. Cox. GD-Wheel: a cost-aware replacement policy for key-value stores. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys'15, pages 1–15, Bordeaux France, April 2015. ACM.

[121] Jinhong Li, Qiuping Wang, Patrick PC Lee, and Chao Shi. An in-depth analysis of cloud block storage workloads in large-scale production. In *2020 IEEE International Symposium on Workload Characterization (IISWC)*, pages 37–47. IEEE, 2020.

[122] Yuliang Li, Gautam Kumar, Hema Hariharan, Hassan Wassel, Peter Hochschild, Dave Platt, Simon Sabato, Minlan Yu, Nandita Dukkipati, Prashant Chandra, et al. Sundial: Fault-tolerant clock synchronization for datacenters. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, pages 1171–1186, USA, 2020. USENIX Association.

[123] Chieh-Jan Mike Liang, Zilin Fang, Yuqing Xie, Fan Yang, Zhao Lucis Li, Li Lyna Zhang, Mao Yang, and Lidong Zhou. On modular learning of distributed systems for predicting {End-to-End} latency. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1081–1095, USA, 2023. usenix.

[124] Lightstep. Opentelemetry: Best practices on sampling. Retrieved December 2020 from `https://opentelemetry.lightstep.com/best-practices/sampling/`, 2020.

[125] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: A holistic approach to fast In-Memory Key-Value storage. In *11th USENIX symposium on networked systems design and implementation*, NSDI'14, pages 429–444, Seattle, WA, April 2014. USENIX Association.

[126] Ping Liu, Haowen Xu, Qianyu Ouyang, Rui Jiao, Zhekang Chen, Shenglin Zhang, Jiahai Yang, Linlin Mo, Jice Zeng, Wenman Xue, et al. Unsupervised detection of microservice trace anomalies through service-level deep Bayesian networks. In *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, pages 48–58, USA, 2020. IEEE, IEEE.

[127] Marten Lohstroh, Christian Menard, Soroush Bateni, and Edward A Lee. Toward a lingua franca for deterministic concurrent systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 20(4):1–27, 2021.

[128] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. Characterizing microservice dependency and performance: Alibaba trace analysis. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 412–426, New York, NY, USA, 2021. Association for Computing Machinery.

[129] Shutian Luo, Huanle Xu, Kejiang Ye, Guoyao Xu, Liping Zhang, Guodong Yang, and Chengzhong Xu. The power of prediction: microservice auto scaling via workload learning. In *Proceedings of the 13th Symposium on Cloud Computing*, pages 355–369, New York, NY, USA, 2022. Association for Computing Machinery.

[130] Jonathan Mace, Peter Bodik, Rodrigo Fonseca, and Madanlal Musuvathi. Retro: Targeted Resource Management in Multi-Tenant Distributed Systems. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15)*, USA, 2015. usenix.

[131] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems. In *25th ACM Symposium on Operating Systems Principles (SOSP'15)*, New York, NY, USA, 2015. Association for Computing Machinery.

[132] Bruce M. Maggs and Ramesh K. Sitaraman. Algorithmic Nuggets in Content Delivery. *ACM SIGCOMM Computer Communication Review*, 45(3):52–66, July 2015.

[133] Bruce M Maggs and Ramesh K Sitaraman. Algorithmic nuggets in content delivery. *ACM SIGCOMM Computer Communication Review*, 45(3):52–66, 2015.

[134] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, and Saurabh Bagchi. ORION and the three rights: Sizing,

bundling, and prewarming for serverless DAGs. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 303–320, Carlsbad, CA, July 2022. USENIX Association.

[135] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Eshaan Minocha, Sameh Elnikety, Saurabh Bagchi, and Somali Chaterji. Wisefuse: Workload characterization and dag transformation for serverless workflows. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 6(2):1–28, 2022.

[136] Adnan Maruf, Ashikee Ghosh, Janki Bhimani, Daniel Campello, Andy Rudoff, and Raju Rangaswami. MULTI-CLOCK: Dynamic Tiering for Hybrid Memory Systems. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 925–937, April 2022. ISSN: 2378-203X.

[137] Sajee Mathew and J Varia. Overview of Amazon Web Services. *Amazon Whitepapers*, 2014.

[138] Sara McAllister, Benjamin Berg, Julian Tutuncu-Macias, Juncheng Yang, Sathya Gunasekar, Jimmy Lu, Daniel S. Berger, Nathan Beckmann, and Gregory R. Ganger. Kangaroo: Theory and practice of caching billions of tiny objects on flash. In *ACM Transactions on Storage*, volume 18 of *TOS'22*, August 2022.

[139] Nimrod Megiddo and Dharmendra S Modha. ARC: A self-tuning, low overhead replacement cache. In *2nd USENIX conference on file and storage technologies*, FAST'03, 2003.

[140] Sailesh Mukil. Cache warming: Leveraging ebs for moving petabytes of data. `https://netflixtechblog.medium.com/cache-warming-leveraging-ebs-for-moving-petabytes-of-data-adcf7a4a78c3`. Accessed: 2023-04-27.

[141] Ali Najafi and Michael Wei. Graham: Synchronizing clocks by leveraging local clock properties. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 453–466, USA, 2022. USENIX Association.

[142] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. MSR Cambridge traces (SNIA IOTTA trace set 388). In Geoff Kuenning, editor, *SNIA IOTTA Trace Repository*. Storage Networking Industry Association, March 2007.

[143] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. Write off-loading: Practical power management for enterprise storage. In *6th USENIX Conference on File and Storage Technologies (FAST 08)*, San Jose, CA, February 2008. USENIX Association.

[144] Usama Naseer and Theophilus A Benson. Configanator: A data-driven approach to improving {CDN} performance. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1135–1158, 2022.

[145] Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and Hari Balakrishnan. Mahimahi: accurate {Record-and-Replay} for {HTTP}. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 417–429, 2015.

[146] Nginx. Nginx. `https://nginx.org/`. Accessed: 2023-02-06.

[147] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, and others. Scaling memcache at facebook. In *10th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'13, pages 385–398, 2013.

[148] Erik Nygren, Ramesh K. Sitaraman, and Jennifer Sun. The Akamai network: a platform for high-performance internet applications. *ACM SIGOPS Operating Systems Review*, 44(3):2–19, August 2010.

[149] Oleg Obleukhov. Building a more accurate time service at facebook scale. https://engineering.fb.com/2020/03/18/production-engineering/ntp-service/, 2020. Accessed: 2023-05-30.

[150] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The LRU-K page replacement algorithm for database disk buffering. *ACM SIGMOD Record*, 22(2):297–306, June 1993.

[151] OpenTelemetry: An Observability Framework for Cloud-Native Software. http://opentelemetry.io/, 2023. Accessed: 2023-05-21.

[152] OpenTracing: Vendor-Neutral APIs and Instrumentation for Distributed Tracing. http://opentracing.io/, 2023. Accessed: 2023-05-21.

[153] Krzysztof Ostrowski, Gideon Mann, and Mark Sandler. Diagnosing latency in multi-tier black-box services. In *4th International Workshop on Large-Scale Distributed Systems and Middleware (LADIS'11)*, USA, 2011.

[154] Maulik Pandey. Building Netflix's Distributed Tracing Infrastructure. Retrieved December 2022 from https://netflixtechblog.com/building-netflixs-distributed-tracing-infrastructure-bb856c319304, 2019.

[155] Sejin Park and Chanik Park. FRD: A filtering based buffer cache algorithm that considers both frequency and reuse distance. In *Proc. of the 33rd IEEE International Conference on Massive Storage Systems and Technology (MSST)*, 2017.

[156] Austin Parker, Daniel Spoonhower, Jonathan Mace, Ben Sigelman, and Rebecca Isaacs. *Distributed Tracing in Practice: Instrumenting, Analyzing, and Debugging Microservices.* O'Reilly Media, USA, 2020.

[157] Tirthak Patel and Devesh Tiwari. Clite: Efficient and qos-aware co-location of multiple latency-critical jobs for warehouse scale computers. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 193–206. IEEE, 2020.

[158] Judea Pearl. Causal inference in statistics: An overview. *Statistics surveys*, 3:96–146, 2009.

[159] Judea Pearl. The seven tools of causal inference, with reflections on machine learning. *Communications of the ACM*, 62(3):54–60, 2019.

[160] Judea Pearl and Dana Mackenzie. *The Book of Why: the new science of cause and effect*. Basic Books, USA, 2018.

[161] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. FIRM: An intelligent fine-grained resource management framework for SLO-oriented microservices. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 805–825, USA, November 2020. USENIX Association.

[162] Ziyue Qiu, Juncheng Yang, Juncheng Zhang, Cheng Li, Xiaosong Ma, Qi Chen, Mao Yang, and Yinlong Xu. Frozenhot cache: Rethinking cache management for modern software. In *Twenty-third EuroSys Conference*, EuroSys'23, New York, NY, USA, 2023. Association for Computing Machinery.

[163] Joy Rahman and Palden Lama. Predicting the end-to-end tail latency of containerized microservices in the cloud. In *2019 IEEE International Conference on Cloud Engineering (IC2E)*, pages 200–210, USA, 2019. IEEE, IEEE.

[164] KV Rashmi, Mosharaf Chowdhury, Jack Kosaian, Ion Stoica, and Kannan Ramchandran. EC-Cache:load-balanced,low-latency cluster caching with online

erasure coding. In *12th USENIX symposium on operating systems design and implementation*, OSDI'16, pages 401–417, 2016.

[165] Lenin Ravindranath, Jitendra Padhye, Sharad Agarwal, Ratul Mahajan, Ian Obermiller, and Shahin Shayandeh. Appinsight: Mobile app performance monitoring in the wild. In *Presented as part of the 10th {USENIX} symposium on operating systems design and implementation ({OSDI} 12)*, pages 107–120, USA, 2012. USENIX Association.

[166] Lenin Ravindranath, Jitendra Padhye, Ratul Mahajan, and Hari Balakrishnan. Timecard: Controlling user-perceived delays in server-based mobile applications. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 85–100, New York, NY, USA, 2013. Association for Computing Machinery.

[167] Patrick Reynolds, Charles Edwin Killian, Janet L Wiener, Jeffrey C Mogul, Mehul A Shah, and Amin Vahdat. Pip: Detecting the unexpected in distributed systems. In *USENIX Symposium on the Networked Systems Design and Implemntation (NSDI'06)*, volume 6, pages 9–9, USA, 2006. USENIX association.

[168] John T. Robinson and Murthy V. Devarakonda. Data cache management using frequency-based replacement. In *Proceedings of the 1990 ACM SIGMETRICS conference on measurement and modeling of computer systems*, SIGMETRICS'90, pages 134–142, New York, NY, USA, 1990. Association for Computing Machinery.

[169] Emanuele Rocca. Wikimedia's cdn up to 2018: Varnish and ipsec. `https://techblog.wikimedia.org/2020/10/14/wikimedias-cdn/`, 2020. Accessed: 2024-01-17.

[170] Liana V. Rodriguez, Farzana Yusuf, Steven Lyons, Eysler Paz, Raju Rangaswami, Jason Liu, Ming Zhao, and Giri Narasimhan. Learning Cache Replacement with

CACHEUS. In *19th USENIX Conference on File and Storage Technologies*, FAST'21, pages 341–354. USENIX Association, February 2021.

[171] Andreas Rogge-Solti, Wil MP van der Aalst, and Mathias Weske. Discovering stochastic petri nets with arbitrary delay distributions from event logs. In *Business Process Management Workshops: BPM 2013 International Workshops, Beijing, China, August 26, 2013, Revised Papers 11*, pages 15–27. Springer, 2014.

[172] Cynthia Rudin. Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead. *Nature machine intelligence*, 1(5):206–215, 2019.

[173] Ruslan Meshenberg, Josh Evan. Netflix at aws re:invent 2015. `https://netflixtechblog.com/netflix-at-aws-re-invent-2015-2bc50551dead`, 2015. Accessed: 2023-05-30.

[174] Raja R. Sambasivan, Alice X. Zheng, Michael De Rosa, Elie Krevat, Spencer Whitman, Michael Stroucken, William Wang, Lianghong Xu, and Gregory R. Ganger. Diagnosing performance changes by comparing request flows. In *NSDI'11: Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, 2011.

[175] Justin Sheehy. There is no now. *Commun. ACM*, 58(5):36–41, apr 2015.

[176] Yuri Shkoro. Conquering Microservices Complexity at Uber with Distributed Tracing (presentation). Received June 2021 from `https://www.infoq.com/presentations/uber-microservices-distributed-tracing/`, 2019.

[177] Yuri Shkuro. *Mastering Distributed Tracing*. Packt Publishing, USA, Feb 2019.

[178] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010.

[179] Arjun Singhvi, Aditya Akella, Maggie Anderson, Rob Cauble, Harshad Deshmukh, Dan Gibson, Milo M. K. Martin, Amanda Strominger, Thomas F. Wenisch, and Amin Vahdat. CliqueMap: productionizing an RMA-based distributed caching system. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM'21, pages 93–105, Virtual Event USA, August 2021. ACM.

[180] Yannis Smaragdakis, Scott Kaplan, and Paul Wilson. EELRU: simple and effective adaptive page replacement. *ACM SIGMETRICS Performance Evaluation Review*, 27(1):122–133, May 1999.

[181] Alan Jay Smith. Sequentiality and prefetching in database systems. *ACM Transactions on Database Systems*, 3(3):223–247, September 1978.

[182] Zhenyu Song, Daniel S Berger, Kai Li, Anees Shaikh, Wyatt Lloyd, Soudeh Ghorbani, Changhoon Kim, Aditya Akella, Arvind Krishnamurthy, Emmett Witchel, and others. Learning relaxed belady for content distribution network caching. In *17th USENIX symposium on networked systems design and implementation*, NSDI'20, pages 529–544, 2020.

[183] Zhenyu Song, Kevin Chen, Nikhil Sarda, Deniz Altinbuken, Eugene Brevdo, Jimmy Coleman, Xiao Ju, Pawel Jurczyk, Richard Schooler, and Ramki Gummadi. Halp: Heuristic aided learned preference eviction policy for youtube content delivery network. In *20th USENIX Symposium on Networked Systems Design and Implementation*, pages 1149–1163, Boston, MA, April 2023. USENIX Association.

[184] Zhenyu Song, Kevin Chen, Nikhil Sarda, Deniz Altınbüken, Eugene Brevdo, Jimmy Coleman, Xiao Ju, Pawel Jurczyk, Richard Schooler, and Ramki Gummadi. {HALP}: Heuristic aided learned preference eviction policy for {YouTube} content delivery network. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1149–1163, 2023.

[185] Zhenyu Song, Kevin Chen, ikhil Sarda, Deniz Altınbüken, Eugene Brevdo, Jimmy Coleman, Xiao Ju, Pawel Jurczyk, Richard Schooler, and Ramki Gummadi. HALP: Heuristic Aided Learned Preference Eviction Policy for YouTube Content Delivery Network. In *20th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'23, pages 1149–1163, 2023.

[186] Benjamin Speich, Belinda von Niederhäusern, Nadine Schur, Lars G Hemkens, Thomas Fürst, Neera Bhatnagar, Reem Alturki, Arnav Agarwal, Benjamin Kasenda, Christiane Pauli-Magnus, et al. Systematic review on costs and resource use of randomized clinical trials shows a lack of transparent and comprehensive data. *Journal of clinical epidemiology*, 96:1–11, 2018.

[187] Kunwadee Sripanidkulchai, Bruce Maggs, and Hui Zhang. An analysis of live streaming workloads on the internet. In *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, pages 41–54, 2004.

[188] Konstantinos Stamos, George Pallis, Athena Vakali, Dimitrios Katsaros, Antonis Sidiropoulos, and Yannis Manolopoulos. CDNsim: A simulation tool for content distribution networks. *ACM Transactions on Modeling and Computer Simulation*, 20(2):10:1–10:40, May 2010.

[189] Volker Stocker. Interconnection and capacity allocation for all-ip networks: walled gardens or full integration? TPRC, 2015.

[190] Volker Stocker, Georgios Smaragdakis, William Lehr, and Steven Bauer. The

growing complexity of content delivery networks: Challenges and implications for the internet ecosystem. *Telecommunications Policy*, 41(10):1003–1016, 2017.

[191] Aditya Sundarrajan, Mingdong Feng, Mangesh Kasbekar, and Ramesh K Sitaraman. Footprint descriptors: Theory and practice of cache provisioning in a global cdn. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*, pages 55–67, 2017.

[192] Linpeng Tang, Qi Huang, Wyatt Lloyd, Sanjeev Kumar, and Kai Li. RIPQ: Advanced photo caching on flash for facebook. In *13th USENIX Conference on File and Storage Technologies*, FAST'15, pages 373–386, 2015.

[193] Mukarram Tariq, Amgad Zeitoun, Vytautas Valancius, Nick Feamster, and Mostafa Ammar. Answering what-if deployment and configuration questions with wise. In *Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, pages 99–110, New York, NY, USA, 2008. Association for Computing Machinery.

[194] Arash Tavakkol, Juan Gómez-Luna, Mohammad Sadrosadati, Saugata Ghose, and Onur Mutlu. {MQSim}: A framework for enabling realistic studies of modern {Multi-Queue}{SSD} devices. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 49–66, 2018.

[195] Parth Thakkar, Rohan Saxena, and Venkata N Padmanabhan. AutoSens: inferring latency sensitivity of user activity through natural experiments. In *Proceedings of the 21st ACM Internet Measurement Conference (IMC'21)*, pages 15–21, New York, NY, USA, 2021. Association for Computing Machinery.

[196] Sudhir Tonse. Scalable microservices at netflix. challenges and tools of the trade. `https://www.infoq.com/presentations/netflix-ipc/`, 2015. Accessed: 2023-05-21.

[197] Mert Toslali, Emre Ates, Alex Ellis, Zhaoqi Zhang, Darby Huye, Lan Liu, Samantha Puterman, Ayse K. Coskun, and Raja R. Sambasivan. Automating instrumentation choices for performance problems in distributed applications with VAIF. In *SoCC '21: Proceedings of the Twelfth Symposium on Cloud Computing*, 2021.

[198] Varnish. Varnish cache. `https://varnish-cache.org/`. Accessed: 2023-02-06.

[199] Kaushik Veeraraghavan, Justin Meza, David Chou, Wonho Kim, Sonia Margulis, Scott Michelson, Rajesh Nishtala, Daniel Obenshain, Dmitri Perelman, and Yee Jiun Song. Kraken: Leveraging live traffic tests to identify and resolve resource utilization bottlenecks in large scale web services. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 635–651, USA, 2016. USENIX Association.

[200] Kaushik Veeraraghavan, Justin Meza, Scott Michelson, Sankaralingam Panneerselvam, Alex Gyori, David Chou, Sonia Margulis, Daniel Obenshain, Shruti Padmanabha, Ashish Shah, et al. Maelstrom: Mitigating datacenter-level disasters by draining interdependent traffic safely and efficiently. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 373–389, USA, 2018. USENIX association.

[201] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. Ernest: Efficient performance prediction for large-scale advanced analytics. In *13th {USENIX} symposium on networked systems design and implementation ({NSDI} 16)*, pages 363–378, USA, 2016. USENIX Association.

[202] Giuseppe Vietri, Liana V. Rodriguez, Wendy A. Martinez, Steven Lyons, Jason Liu, Raju Rangaswami, Ming Zhao, and Giri Narasimhan. Driving cache

replacement with ML-based LeCaR. In *10th USENIX workshop on hot topics in storage and file systems*, hotStorage'18, Boston, MA, July 2018. USENIX Association.

[203] Carl Waldspurger, Trausti Saemundsson, Irfan Ahmad, and Nohhyun Park. Cache modeling and optimization using miniature simulations. In *2017 USENIX annual technical conference*, ATC'17, pages 487–498, Santa Clara, CA, July 2017. USENIX Association.

[204] Carl A. Waldspurger, Nohhyun Park, Alexander Garthwaite, and Irfan Ahmad. Efficient MRC construction with SHARDS. In *13th USENIX conference on file and storage technologies*, FAST'15, pages 95–110, Santa Clara, CA, February 2015. USENIX Association.

[205] Hua Wang, Xinbo Yi, Ping Huang, Bin Cheng, and Ke Zhou. Efficient SSD Caching by Avoiding Unnecessary Writes using Machine Learning. In *Proceedings of the 47th International Conference on Parallel Processing*, ICPP'18, pages 1–10, Eugene OR USA, August 2018. ACM.

[206] Qiuping Wang, Jinhong Li, Tao Ouyang, Chao Shi, and Lilong Huang. Separating data via block invalidation time inference for write amplification reduction in {Log-Structured} storage. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 429–444, 2022.

[207] Alec Warner and Štěpán Davidovič. The site reliability engineering workbook chapter: Canarying releases. 2018.

[208] wikimedia. Analytics/data lake/traffic/caching. `https://wikitech.wikimedia.org/wiki/Analytics/Data_Lake/Traffic/Caching`. Accessed: 2023-02-06.

[209] Simon Wistow. Why having more pops isn't always better. https://www.fastly.com/blog/why-having-more-pops-isnt-always-better, 2016. Accessed: 2024-03-13.

[210] Theodore M Wong and John Wilkes. My cache or yours?: Making storage more exclusive. In *USENIX Annual Technical Conference*, ATC'02, pages 161–175, 2002.

[211] Nan Wu and Pengcheng Li. Phoebe: Reuse-Aware Online Caching with Reinforcement Learning for Emerging Storage Models, November 2020.

[212] Xingbo Wu, Li Zhang, Yandong Wang, Yufei Ren, Michel Hack, and Song Jiang. zExpander: a key-value cache with both high performance and fewer misses. In *Proceedings of the Eleventh European Conference on Computer Systems*, Eurosys'16, pages 1–15, London United Kingdom, April 2016. ACM.

[213] Yang Wu, Ang Chen, and Linh Thi Xuan Phan. Zeno: diagnosing performance problems with temporal provenance. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI'19)*, pages 395–420, USA, 2019. USENIX Association.

[214] Yuchen Wu. Why we started putting unpopular assets in memory. https://blog.cloudflare.com/why-we-started-putting-unpopular-assets-in-memory/. Accessed: 2023-02-06.

[215] Gang Yan and Jian Li. RL-Bélády: A Unified Learning Framework for Content Caching. In *Proceedings of the 28th ACM International Conference on Multimedia*, pages 1009–1017, Seattle WA USA, October 2020. ACM.

[216] Gang Yan and Jian Li. Towards Latency Awareness for Content Delivery Network Caching. ATC'22, pages 789–804, 2022.

[217] C-Q Yang and Barton P Miller. Critical path analysis for the execution of parallel and distributed programs. In *The 8th International Conference on Distributed*, pages 366–367, USA, 1988. IEEE Computer Society, IEEE.

[218] Juncheng Yang. libcachesim: a high-performance library for building cache simulators. `https://github.com/1a1a11a/libcachesim/`. Accessed: 2023-04-27.

[219] Juncheng Yang, Ziming Mao, Yao Yue, and K. V. Rashmi. GL-Cache: Group-level learning for efficient and high-performance caching. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, FAST'23, pages 115–134, 2023.

[220] Juncheng Yang, Ziyue Qiu, Yazhuo Zhang, Yao Yue, and K. V. Rashmi. FIFO can be better than LRU: the power of lazy promotion and quick demotion. In *The 19th Workshop on Hot Topics in Operating Systems (HotOS 23)*, 2023.

[221] Juncheng Yang, Ziyue Qiu, Yazhuo Zhang, Yao Yue, and K. V. Rashmi. FIFO can be Better than LRU: the Power of Lazy Promotion and Quick Demotion. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, HOTOS '23, pages 70–79, New York, NY, USA, June 2023. Association for Computing Machinery.

[222] Juncheng Yang, Ziyue Qiu, Yazhuo Zhang, Yao Yue, and KV Rashmi. Fifo can be better than lru: the power of lazy promotion and quick demotion. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, pages 70–79, 2023.

[223] Juncheng Yang, Anirudh Sabnis, Daniel S. Berger, K. V. Rashmi, and Ramesh K. Sitaraman. C2DN: How to harness erasure codes at the edge for efficient content delivery. In *19th USENIX symposium on networked systems design and*

implementation, NSDI'22, pages 1159–1177, Renton, WA, April 2022. USENIX Association.

[224] Juncheng Yang, Yao Yue, and K. V. Rashmi. Slides of a large scale analysis of hundreds of in-memory cache clusters at twitter. `https://www.usenix.org/sites/default/files/conference/protected-files/osdi20_slides_yang.pdf`. Accessed: 2023-04-27.

[225] Juncheng Yang, Yao Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *14th USENIX symposium on operating systems design and implementation*, OSDI'20, pages 191–208. USENIX Association, November 2020.

[226] Juncheng Yang, Yao Yue, and Rashmi Vinayak. Segcache: a memory-efficient and scalable in-memory key-value cache for small objects. In *18th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'21, pages 503–518. USENIX Association, April 2021.

[227] Juncheng Yang, Yazhuo Zhang, Ziyue Qiu, Yao Yue, and K.V. Rashmi. Fifo queues are all you need for cache eviction. In *Symposium on Operating Systems Principles (SOSP'23)*, 2023.

[228] Tzu-Wei Yang, Seth Pollen, Mustafa Uysal, Arif Merchant, and Homer Wolfmeister. CacheSack: Admission Optimization for Google Datacenter Flash Caches. In *2022 USENIX Annual Technical Conference*, ATC'22, pages 1021–1036, Carlsbad, CA, July 2022. USENIX Association.

[229] Tao Yu, Yue Zhang, and Kwei-Jay Lin. Efficient algorithms for web services selection with end-to-end qos constraints. *ACM Transactions on the Web (TWEB)*, 1(1):6–es, 2007.

[230] Lei Zhang, Vaastav Anand, Zhiqiang Xie, Ymir Vigfusson, and Jonathan Mace. The benefit of hindsight: Tracing edge-cases in distributed systems. In *NSDI'23: Proceedings of the 20th USENIX Conference on Networked Systems Design and Implementation*, pages 321–339, USA, 2023. USENIX association.

[231] Yazhuo Zhang, Rebecca Isaacs, Yao Yue, Juncheng Yang, Lei Zhang, and Ymir Vigfusson. Latenseer: Causal modeling of end-to-end latency distributions by harnessing distributed tracing. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*, pages 502–519, 2023.

[232] Yazhuo Zhang, Juncheng Yang, Yao Yue, Ymir Vigfusson, and KV Rashmi. Sieve is simpler than lru: an efficient turn-key eviction algorithm for web caches. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24). USENIX Association*, 2024.

[233] Yilei Zhang, Zibin Zheng, and Michael R Lyu. Wspred: A time-aware personalized qos prediction framework for web services. In *2011 IEEE 22nd international symposium on software reliability engineering*, pages 210–219. IEEE, 2011.

[234] Yiying Zhang, Gokul Soundararajan, Mark W. Storer, Lakshmi N. Bairavasundaram, Sethuraman Subbiah, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Warming up storage-level caches with bonfire. In *Proceedings of the 11th USENIX conference on File and Storage Technologies*, FAST'13, pages 59–72, USA, February 2013. USENIX Association.

[235] Zhizhou Zhang, Murali Krishna Ramanathan, Prithvi Raj, Abhishek Parwal, Timothy Sherwood, and Milind Chabbi. CRISP: Critical path analysis of large-scale microservice architectures. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 655–672, USA, 2022. USENIX association.

[236] Chenxingyu Zhao, Tapan Chugh, Jaehong Min, Ming Liu, and Arvind Krishna-

murthy. Dremel: Adaptive configuration tuning of rocksdb kv-store. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 6(2):1–30, 2022.

[237] Chen Zhong, Xingsheng Zhao, and Song Jiang. LIRS2: an improved LIRS replacement algorithm. In *Proceedings of the 14th ACM International Conference on Systems and Storage*, SYSTOR'21, pages 1–12, Haifa Israel, June 2021. ACM.

[238] Hao Zhou, Ming Chen, Qian Lin, Yong Wang, Xiaobin She, Sifan Liu, Rui Gu, Beng Chin Ooi, and Junfeng Yang. Overload control for scaling wechat microservices. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 149–161, New York, NY, USA, 2018. Association for Computing Machinery.

[239] Ke Zhou, Si Sun, Hua Wang, Ping Huang, Xubin He, Rui Lan, Wenyan Li, Wenji Liu, and Tianming Yang. Tencent photo cache traces (SNIA IOTTA trace set 27476). In Geoff Kuenning, editor, *SNIA IOTTA Trace Repository*. Storage Networking Industry Association, February 2016.

[240] Ke Zhou, Si Sun, Hua Wang, Ping Huang, Xubin He, Rui Lan, Wenyan Li, Wenjie Liu, and Tianming Yang. Demystifying cache policies for photo stores at scale: A tencent case study. In *Proceedings of the 2018 International Conference on Supercomputing*, ICS '18, page 284–294, New York, NY, USA, 2018. Association for Computing Machinery.

[241] Ke Zhou, Si Sun, Hua Wang, Ping Huang, Xubin He, Rui Lan, Wenyan Li, Wenjie Liu, and Tianming Yang. Demystifying Cache Policies for Photo Stores at Scale: A Tencent Case Study. In *Proceedings of the 2018 International Conference on Supercomputing*, pages 284–294, Beijing China, June 2018. ACM.

[242] Y. Zhou, Z. Chen, and K. Li. Second-level buffer cache management. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):505–519, June 2004.

[243] Yuanyuan Zhou, James Philbin, and Kai Li. The multi-queue replacement algorithm for second level buffer caches. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATC'01, pages 91–104, USA, 2001. USENIX Association.

[244] Zipkin: A Distributed Tracing System. `http://zipkin.io/`, 2023. Accessed: 2023-05-21.

[245] Behrouz Zolfaghari, Gautam Srivastava, Swapnoneel Roy, Hamid R. Nemati, Fatemeh Afghah, Takeshi Koshiba, Abolfazl Razi, Khodakhast Bibak, Pinaki Mitra, and Brijesh Kumar Rai. Content Delivery Networks: State of the Art, Trends, and Future Roadmap. *ACM Computing Surveys*, 53(2):34:1–34:34, April 2020.