RobinHood: Tail Latency-Aware Caching — Dynamically Reallocating from Cache-Rich to Cache-Poor

Daniel S. Berger[1], Benjamin Berg[1], Timothy Zhu[2], Mor Harchol-Balter[1], and Siddhartha Sen[3]

[1]Carnegie Mellon University — [2]Penn State — [3]Microsoft Research

Abstract

Tail latency is of great importance in user-facing web services. However, maintaining low tail latency is challenging, because a single request to a web application server results in multiple queries to complex, diverse backend services (databases, recommender systems, ad systems, etc.). A *request* is not complete until *all of its queries* have completed. We analyze a Microsoft production system and find that backend query latencies vary by more than two orders of magnitude *across backends* and *over time*, resulting in high request tail latencies.

We propose a novel solution for maintaining low request tail latency: repurpose existing caches to mitigate the effects of backend latency variability, rather than just caching popular data. Our solution, RobinHood, dynamically reallocates cache resources from the cache-rich (backends which don't affect request tail latency) to the cache-poor (backends which affect request tail latency). We evaluate RobinHood with production traces on a 50-server cluster with 20 different backend systems. Surprisingly, we find that RobinHood can directly address tail latency even if working sets are much larger than the cache size. In the presence of load spikes, RobinHood meets a 150ms P99 goal 99.7% of the time, whereas the next best policy meets this goal only 70% of the time.

1 Introduction

Request tail latency matters. Providers of large user-facing web services have long faced the challenge of achieving low request latency. Specifically, companies are interested in maintaining low *tail latency*, such as the 99th percentile (P99) of request latencies [26, 27, 36, 44, 63, 83, 92]. Maintaining low tail latencies in real-world systems is especially difficult when incoming *requests* are complex, consisting of multiple *queries* [4, 26, 36, 90], as is common in *multitier architectures*. Figure 1 shows an example of a multitier architecture: each user request is received by an application server, which then sends queries to the necessary *backends*, waits until all queries have completed, and then packages the results for delivery back to the user. Many large web services, such as Wikipedia [71], Amazon [27], Facebook [20], Google [26] and Microsoft, use this design pattern.

The queries generated by a single request are independently processed in parallel, and may be spread over many

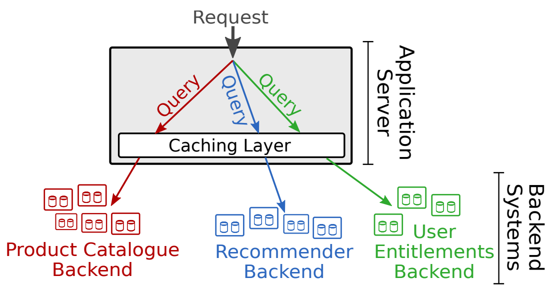


Figure 1: In a multitier system, users submit individual *requests*, which are received by application servers. To complete a request, an application server issues a series of *queries* to various *backend services*. The request is only complete when all of its queries have completed.

backend services. Since each request must wait for all of its queries to complete, the overall *request latency* is defined to be the latency of the request's *slowest* query. Even if almost all backends have low tail latencies, the tail latency of the *maximum* of several queries could be high.

For example, consider a stream of requests where each request queries a single backend 10 times in parallel. Each request's latency is equal to the *maximum* of its ten queries, and could therefore greatly exceed the P99 query latency of the backend. The P99 request latency in this case actually depends on a higher percentile of backend query latency [26]. Unfortunately, as the number of backends in the system increases and the workload becomes more heterogeneous, P99 request latency may depend on different (higher or lower) percentiles of query latency for each backend, and determining what these important percentiles are is difficult.

To illustrate this complexity, this paper focuses on a concrete example of a large multitier architecture: the OneRF page rendering framework at Microsoft. OneRF serves a wide range of content including news (msn.com) and online retail software stores (microsoft.com, xbox.com). It relies on more than 20 backend systems, such as product catalogues, recommender systems, and user entitlement systems (Figure 1).

The source of tail latency is dynamic. It is common in multitier architectures that the particular backend causing high request latencies changes over time. For example,

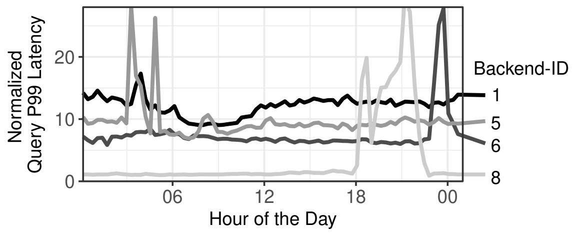


Figure 2: Normalized 99-th percentile (P99) latencies over the course of a typical day for four backends in the OneRF production system. Each backend has the highest tail latency among all backends at some point during the day, indicating that latencies are not only unbalanced between backends, but the imbalance changes over time.

Figure 2 shows the P99 query latency in four typical backend services from OneRF over the course of a typical day. Each of the four backends at some point experiences high P99 query latency, and is thus responsible for some high-latency requests. However, this point happens at a different time for each backend. Thus any mechanism for identifying the backends that affect tail request latency should be dynamic—accounting for the fact that the latency profile of each backend changes over time.

Existing approaches. Some existing approaches for reducing tail latency rely on load balancing between servers and aim to equalize query tail latencies between servers. This should reduce the maximum latency across multiple queries. Unfortunately, the freedom to load balance is heavily constrained in a multitier architecture, where a given backend typically is unable to answer a query originally intended for another backend system (e.g., the user entitlements backend cannot answer product catalogue queries). While some limited load balancing can be done between replicas of a single backend system, load balancing is impossible across *different* backends.

Alternatively, one might consider reducing tail latency by dynamically auto-scaling backend systems—temporarily allocating additional servers to the backends currently experiencing high latency. Given the rapid changes in latency shown in Figure 2, it is important to be able to scale backends quickly. Unfortunately, dynamic auto-scaling is difficult to do quickly in multitier systems like OneRF because backends are stateful [30]. In fact, many of the backends at Microsoft do some form of auto-scaling, and Figure 2 shows that the systems are still affected by latency spikes.

The RobinHood solution. In light of these challenges, we suggest a novel idea for minimizing request tail latency that is agnostic to the design and functionality of the backend services. We propose repurposing the existing caching layer (see Figure 1) in the multitier system to directly address request tail latency by dynamically partitioning the cache. Our solution, *RobinHood*, dynamically allocates cache space to those backends responsible for high request tail latency (cache-poor backends), while stealing space from backends that do not affect the request tail latency (cache-rich backends). In doing so, RobinHood makes compromises that may seem counterintuitive (e.g., significantly increasing the tail latencies of certain backends) but which ultimately improve overall request tail latency. Since many multitier systems already incorporate a caching layer that is capable of dynamic partitioning, RobinHood can be deployed with very little additional overhead or complexity.

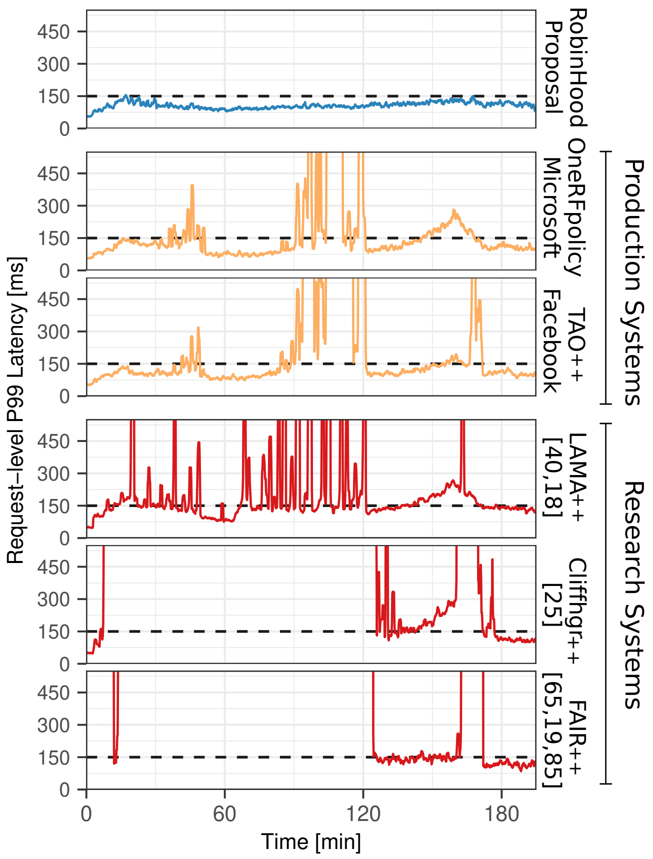


Figure 3: Comparison of the P99 request latency of RobinHood, two production caching systems, and three state-of-the-art research caching systems, which we emulated in our testbed. All systems are subjected to three load spikes, as in Figure 2. We draw a dashed line at 150ms, which is the worst latency under RobinHood.

RobinHood is not a traditional caching system. While many multitier systems employ a caching layer, these caches are often designed only to improve *average* (not tail) latency of *individual queries* (not requests) [3, 11, 17, 25, 40]. In all of the production workloads we study, an application's working set is larger than the available cache space, and thus the caching layer can improve average query latency by allowing fast accesses (cache hits) to the most popular data. By contrast, request tail latency is caused almost entirely by cache misses. In fact, conventional wisdom says that when the application's working set does not fit entirely in the cache, the caching layer does not directly address tail latency [26]. Thus, despite various efforts to optimize the caching layer in both industry and academia (see Section 7), none of these systems are designed to reduce request tail latency.

Contributions. RobinHood is the first caching system that minimizes the request tail latency. RobinHood is driven by a lightweight cache controller that leverages existing caching infrastructure and is agnostic to the design and functionality of the backend systems.

We implement[1] and extensively evaluate the RobinHood system along with several research and production caching systems. Our 50-server testbed includes 20 backend systems that are modeled after the 20 most queried backends from Microsoft's OneRF production system. Figure 3 shows a preview of an experiment where we mimic the backend latency spikes in OneRF: RobinHood meets a 150ms P99 goal 99.7% of the time, whereas the next best policy meets this goal only 70% of the time.

Our contributions are the following:

- **Section 2.** We find that there are many different types of requests, each with their own *request structure* that defines which backend systems are queried. We analyze structured requests within the OneRF production system, and conclude that request structure must be incorporated by any caching system seeking to minimize request tail latency.
- **Section 3.** We present RobinHood, a dynamic caching system which aims to minimize request tail latency by considering request structure. RobinHood identifies which backends contribute to the tail over time, using a novel metric called request blocking count (RBC).
- **Section 4.** We implement RobinHood as a scalable distributed system. We also implement the first distributed versions of state-of-the-art research systems: LAMA [40], Cliffhanger [25], and FAIR [19, 65, 85] to use for comparison.
- **Section 5.** We evaluate RobinHood and prior systems against simultaneous latency spikes across multiple backends, and show that RobinHood is far more robust while imposing negligible overhead.

We discuss how to generalize RobinHood to architecture beyond OneRF in **Section 6**, survey the related work in **Section 7**, and conclude in **Section 8**.

2 Background and Challenges

The RobinHood caching system targets tail latency in multitier architectures, where requests depend on queries to many backends. One such system, the OneRF system, serves several Microsoft storefront properties and relies on a variety of backend systems. Each OneRF application server has a *local* cache. Incoming requests are split into queries which first lookup up in the cache. Cache misses are then forwarded, in parallel, to clusters of backend servers. Once each query has been answered, the application server can serve the user request. Thus, each request takes as long as its slowest query. A OneRF request can send any number of queries (including 0) to each backend system.

Before we describe the RobinHood algorithm in Section 3, we discuss the goal of RobinHood in more depth, and how prior caching systems fail to achieve this goal.

2.1 The goal of RobinHood

The key idea behind RobinHood is to identify backends whose queries are responsible for high P99 request latency, which we call "cache-poor" backends. RobinHood then shifts cache resources from the other "cache-rich" backends to the cache-poor backends. RobinHood is a departure from "fair" caching approaches [65], treating queries to cache-rich backends unfairly as they do not affect the P99 request latency. For example, increasing the latency of a query that occurs in parallel with another, longer query, will not increase the *request* latency. By sacrificing the performance of cache-rich backends, RobinHood frees up cache space.

RobinHood allocates free cache space to cache-poor backends (see Section 3). Additional cache space typically improves the hit ratios of these backends, as the working sets of web workloads do not fit into most caches [3, 20, 41, 61, 72]. As the hit ratio increases, fewer queries are sent to the cache-poor backends. Since backend query latency is highly variable in practice [4, 26, 29, 36, 45, 62, 68, 83], decreasing the number of queries to a backend will decrease the number of high-latency queries observed. This will in turn improve the P99 request latency.

In addition, sending fewer queries can also reduce resource congestion and competition in the backends, which is often the cause of high tail latency [26, 35, 77]. Small reductions in resource congestion can have an outsized impact on backend latency [34, 39] and thus significantly improve the request P99 (as we will see in Section 5).

2.2 Challenges of caching for tail latency

We analyze OneRF traces collected over a 24 hour period in March 2018 in a datacenter on the US east coast. The traces contain requests, their queries, and the query latencies for the 20 most queried backend systems, which account for more than 99.95% of all queries.

We identify three key obstacles in using a caching system to minimizing tail request latencies.

[1]RobinHood's source code is available at `https://github.com/dasebe/robinhoodcache`.

2.2.1 Time-varying latency imbalance

As shown in Figure 2, it is common for the latencies of different backends to vary widely. Figure 5 shows that the latency across the 20 backends varies by more than 60×. The fundamental reason for this latency imbalance is that several of these backend systems are complex, distributed systems in their own right. They serve multiple customers within the company, not just OneRF.

In addition to high latency imbalance, backend latencies also change over time (see Figure 2). These changes are frequently caused by customers other than OneRF and thus occur independently of the request stream seen by OneRF applications servers.

Why latency imbalance is challenging for existing systems. Most existing caching systems implicitly assume that latency is balanced. They focus on optimizing cache-centric metrics (e.g., hit ratio), which can be a poor representation of overall performance if latency is imbalanced. For example, a common approach is to partition the cache in order to provide fairness guarantees about the hit ratios of queries to different backends [19, 65, 85]. This approach is represented by the FAIR policy in Figure 3, which dynamically partitions the cache to equalize backend cache hit ratios. If latencies are imbalances between the backends, two cache misses to different backends should not be treated equally. FAIR fails to explicitly account for the latency of cache misses and thus may result in high request latency.

Some production systems do use latency-aware static cache allocations, e.g., the "arenas" in Facebook's TAO [20]. However, manually deriving the optimal static allocation is an open problem [20], and even an "optimal" static allocation will become stale as backend latencies vary over time (see Section 5).

2.2.2 Latency is not correlated with specific queries nor with query rate

We find that high latency is not correlated with specific queries as assumed by cost-aware replacement policies [21, 52]. Query latency is also not correlated with a query's popularity (the rate at which the query occurs), but rather reflects a more holistic state of the backend system at some point in time. This is shown in Figure 4 with a scatterplot of a query's popularity and its latency for the four OneRF backends shown in Figure 2 (other backends look similar).

We also find that query latency is typically not correlated with a backend's query rate. For example, the seventh most queried backend receives only about 0.06× as many queries as the most backend, but has 3× the query latency (Figure 5). This is due to the fact that backend systems are used by customers other than OneRF. Even if OneRF's query rate to a backend is low, another service's query stream may be causing high backend latency. Additionally, queries to some backends take inherently

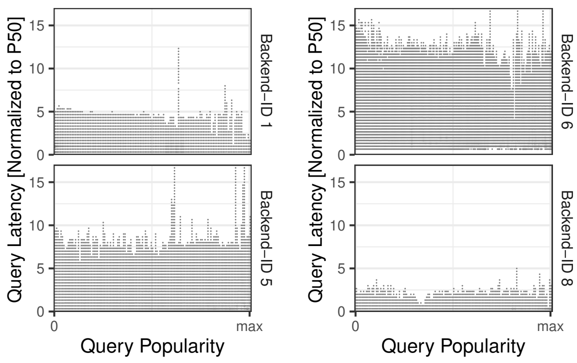


Figure 4: Scatterplots of query popularity and query latency for each backend from Figure 2. We find that query latency is neither correlated with query rate nor with particular queries.

longer than others, e.g., generating a personalized product recommendation takes 4× longer than looking up a catalogue entry.

Why uncorrelated latency is challenging for existing systems. Many caching schemes (including OneRF) share cache space among the backends and use a common eviction policy (such as LRU). Shared caching systems [17, 52] inherently favor backends with higher query rates [9]. Intuitively, this occurs because backends with higher query rates have more opportunities for their objects to be admitted into the cache. Cost-aware replacement policies also suffer from this problem, and are generally ineffective in multitier architectures such OneRF as their assumptions (high latency is correlated with specific queries) are not met.

Another common approach is to partition cache space to maximize overall cache hit ratios as in Cliffhanger [25]. All these approaches allocate cache space in proportion to query rate, which leads to suboptimal cache space allocations when latency is uncorrelated with query rate. As shown in Figure 3, both OneRF and Cliffhanger lead to high P99 request latency. In order to minimize request tail latency, a successful caching policy must directly incorporate backend latency, not just backend query rates.

2.2.3 Latency depends on request structure, which varies greatly

The manner in which an incoming request is split into parallel backend queries by the application server varies between requests. We call the mapping of a request to its component backend queries the *request structure*.

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

Table 1 summarizes how the query traffic of different

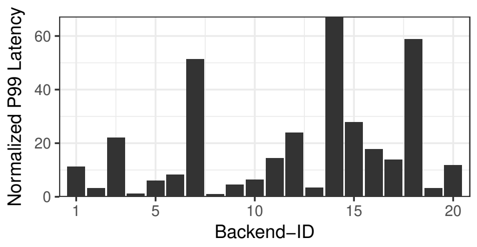


Figure 5: Normalized P99 latencies for the 20 most queried backends in the OneRF system during a typical 10 minute period. The backends are ordered by their query rates during this period. We see that query rate is not correlated with backend tail latency.

backends is affected by the request structure. We list the percentage of the overall number of queries that go to each backend, and the percentage of requests which reference each backend. We also list the average batch size and fanout by backend. We can see that all of these metrics vary across the different backends and are not strongly correlated with each other.

Why request structure poses a challenge for existing systems. There are few caching systems that incorporate latency into their decisions, and they consider the average query latency as opposed to the tail request latency [18, 40]. We find that even after changing these *latency aware* systems to measure the P99 query latency, they remain ineffective (see LAMA++ in Figure 3).

These systems fail because a backend with high query latency does not always cause high request latency. A simple example would be high query latency in backend 14. As backend 14 occurs in less than 0.2% of all requests, its impact on the P99 request latency is limited—even if backend 14 was arbitrarily slow, it could not be responsible for *all* of the requests above the P99 request latency. A scheme that incorporates query rate and latency might decide to allocate most of the cache space towards backend 14, which would not improve the P99 request latency.

While the specific case of backend 14 might be simple to detect, differences in batch sizes and fanout give rise to complicated scenarios. For example, Figure 5 shows that backend 3's query latency is higher than backend 4's query latency. Table 1 shows that, while backend 3 has a large batch size, backend 4 occurs in 4.5× more requests, which makes backend 4 more likely to affect the P99 request latency. In addition, backend 4 occurs in requests with a 55% smaller fanout, which makes it more likely to be the slowest backend, whereas backend 3's query latency is frequently hidden by slow queries to other backends.

As a consequence, minimizing request tail latency is difficult unless request structure is explicitly considered.

Backend-ID	Query %	Request %	Mean Batch Size	Mean Fanout
1	37.7%	14.7%	15.4	5.6
2	16.0%	4.5%	32.3	7.4
3	15.3%	4.5%	25.7	7.4
4	14.0%	20.0%	1.6	4.8
5	7.7%	19.0%	1.9	4.9
6	4.2%	4.7%	14.5	7.3
7	2.4%	10.8%	2.0	5.3
8	1.6%	15.5%	1.0	5.3
9	0.7%	3.4%	2.0	7.5
10	0.2%	0.7%	2.5	9.1

Table 1: Four key metrics describing the 10 most queried OneRF backends. Backend-IDs are ordered by query rate starting with the most queried backend, backend 1. Query % describes the percentage of the total number of queries directed to a given backend. Request % denotes the percentage of requests with at least one query to the given backend. Batch size describes the average number of parallel queries made to the given backend across requests with at least one query to that backend. Fanout describes the average number of backends queried across requests with at least one query to the given backend.

3 The RobinHood Caching System

In this section, we describe the basic RobinHood algorithm (Section 3.1), how we accommodate real-world constraints (Section 3.2), and the high-level architecture of RobinHood (Section 3.3). Implementation details are discussed in Section 4.

3.1 The basic RobinHood algorithm

To reallocate cache space, RobinHood repeatedly taxes every backend by reclaiming 1% of its cache space, identifies which backends are cache-poor, and redistributes wealth to these cache-poor backends.

RobinHood operates over time windows of Δ seconds, where $\Delta = 5$ seconds in our implementation.[2] Within a time window, RobinHood tracks the latency of each request. Since the goal is to minimize the P99 request latency, RobinHood focuses on the set of requests, S, whose request latency is between the P98.5 and P99.5 (we explain this choice of range below). For each request in S, RobinHood tracks the ID of the backend corresponding to the slowest query in the request. RobinHood then counts the number of times each backend produced the slowest query in a request. We call each backend's total count its *request blocking count* (RBC). Backends with a high RBC are frequently the bottleneck in slow requests. RobinHood thus considers a backend's RBC as a measure of how cache-poor it is, and distributes the pooled tax to each backend in proportion to its RBC.

Choosing the RBC metric. The RBC metric captures key aspects of request structure. Recall that, when minimizing request tail latency, it is not sufficient to know

[2]This parameter choice is determined by the time it takes to reallocate 1% of the cache space in off-the-shelf caching systems; see Section 4.

only that a backend produces high query latencies. This backend must also be queried in such a way that it is frequently the slowest backend queried by the slowest requests in the system. Metrics such as batch size and fanout width will determine whether or not a particular backend's latencies are hidden or amplified by the request structure. For example, if a slow backend is queried in parallel with many queries to other backends (high fanout width), the probability of the slow backend producing the slowest query may be relatively small. We would expect this to result in a lower RBC for the slow backend than its query latency might suggest. A backend with a high RBC indicates not only that the backend produced high-latency queries, but that reducing the latency of queries to this backend would have actually reduced the latency of the requests affecting the P99 request latency.

Choosing the set S. The set S is chosen to contain requests whose latency is close to the P99 latency, specifically between the P98.5 and P99.5 latencies. Alternatively, one might consider choosing S to be the set requests with latencies greater than or equal to the P99. However, this set is known to include extreme outliers [27] whose latency, even if reduced significantly, would still be greater than the P99 latency. Improving the latency of such outliers would thus be unlikely to change the P99 request latency. Our experiments indicate that choosing a small interval around the P99 filters out most of these outliers and produces more robust results.

3.2 Refining the RobinHood algorithm

The basic RobinHood algorithm, described above, is designed to directly address the key challenges outlined in Section 2. However, real systems introduce additional complexity that must be addressed. We now describe two additional issues that must be considered to make the RobinHood algorithm effective in practice.

Backends appreciate the loot differently. The basic RobinHood algorithm assumes that redistributed cache space is filled immediately by each backend's queries. In reality, some backends are slow to make use of the additional cache space because their hit ratios are already high. RobinHood detects this phenomenon by monitoring the gap between the allocated and the used cache capacity for each backend. If this gap is more than 30% of the used cache space, RobinHood temporarily ignores the RBC of this backend to avoid wasting cache space. Note that such a backend may continue to affect the request tail latency. RobinHood instead chooses to focus on backends which can make use of additional cache space.

Local decision making and distributed controllers. The basic RobinHood algorithm assumes an abstraction of a single cache with one partition per backend. In reality (e.g., at OneRF), incoming requests are load balanced across a cluster of application servers, each of which has

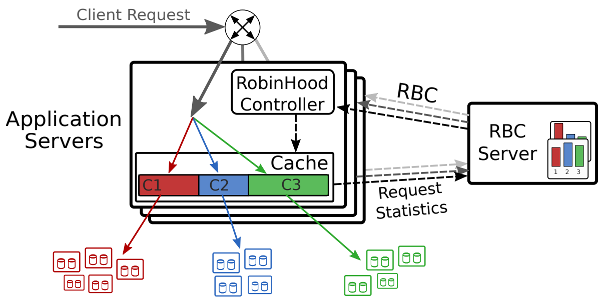


Figure 6: RobinHood adds a distributed controller to each application server and a latency statistics (RBC) server.

its own local cache (see Section 2). Due to random load balancing, two otherwise identical partitions on different application servers may result in different hit ratios[3]. Therefore, additional cache space will be consumed at different rates not only per backend, but also per application server. To account for this, RobinHood's allocation decisions (such as imposing the 30% limit described above) are made locally on each application server. This leads to a distributed controller design, described in Section 3.3.

One might worry that the choice of distributed controllers could lead to diverging allocations and cache space fragmentation across application servers over time. However, as long as two controllers exchange RBC data (see Section 3.3), their cache allocations will quickly converge to the same allocation regardless of initial differences between their allocations. Specifically, given $\Delta = 5$ seconds, any RobinHood cache (e.g., a newly started one) will converge to the average allocation within 30 minutes assuming all servers see sufficient traffic to fill the caches. In other words, the RobinHood cache allocations are *not* in danger of "falling off a cliff" due to diverging allocations — RobinHood's distributed controllers will always push the caches back to the intended allocation within a short time span.

3.3 RobinHood architecture

Figure 6 shows the RobinHood architecture. It consists of application servers and their caches, backend services, and an RBC server.

RobinHood requires a caching system that can be dynamically resized. We use off-the-shelf memcached instances to form the caching layer on each application server in our testbed (see Section 4). Implementing RobinHood requires two additional components not currently used by production systems such as OneRF. First, we add a lightweight cache controller to each application server. The controller implements the RobinHood algorithm (Sections 3.1 and 3.2) and issues resize commands to the local cache's partitions. The input for each controller is the RBC, described in Section 3.1. To prevent

[3]While most caches will contain roughly the same data, it is likely that at least one cache will look notably different from the others.

an all-to-all communication pattern between controllers, we add a centralized RBC server. The RBC server aggregates request latencies from all application servers and computes the RBC for each backend. In our implementation, we modify the application server caching library to send (in batches, every second) each request's latency and the backend ID from the request's longest query. In the OneRF production system, the real-time logging framework already includes all the metrics required to calculate the RBC, so RobinHood does not need to change application libraries. This information already exists in other production systems as well, such as at Facebook [77]. The controllers poll the RBC server for the most recent RBCs each time they run the RobinHood algorithm.

Fault tolerance and scalability. The RobinHood system is robust, lightweight, and scalable. RobinHood controllers are distributed and do not share any state, and RBC servers store only soft state (aggregated RBC from the last one million requests, in a ring buffer). Both components can thus quickly recover after a restart or crash. Just as RobinHood can recover from divergence between cache instances due to randomness (see Section 3.2), RobinHood will recover from any measurement errors that might result in bad reallocation messages being sent to the controllers. The additional components required to run RobinHood (controller and statistics server) are not on the critical path of requests and queries, and thus do not impose any latency overhead. RobinHood imposes negligible overhead and can thus scale to several hundred application servers (Section 5.6).

4 System Implementation and Challenges

To demonstrate the effectiveness and deployability of RobinHood, we implement the RobinHood architecture using an off-the-shelf caching system. In addition, we implement five state-of-the-art caching systems (further described in Section 5.1) on top of this architecture.

4.1 Implementation and testbed

The RobinHood controller is a lightweight Python process that receives RBC information from the global RBC server, computes the desired cache partition sizes, and then issues resize commands to the caching layer. The RBC server and application servers are highly concurrent and implemented in Go. The caching layer is composed of off-the-shelf memcached instances, capable of dynamic resizing via the memcached API. Each application server has a local cache with 32 GB cache capacity.

To test these components, we further implement different types of backend systems and a concurrent traffic generator that sends requests to the application servers. On average, a request to the application server spawns 50 queries. A query is first looked up in the local memcached instance; cache misses are then forwarded to the corresponding backend system. During our experiments the average query rate of the system is 200,000 queries per second (over 500,000 peak). To accommodate this load we had to build highly scalable backend systems. Specifically, we use three different types of backends. A distributed key-value store that performs simple lookup queries (similar to OneRF's product rating and query service). A fast MySQL cluster performs an indexed-join and retrieves data from several columns (similar to OneRF's product catalogue systems). And, a custom-made matrix-multiplier system that imitates a recommendation prediction (similar to various OneRF recommender backends).

Our experimental testbed consists of 16 application servers and 34 backend servers divided among 20 backend services. These components are deployed across 50 Microsoft Azure D16 v3 VMs[4].

4.2 Implementation challenges

The central challenge in implementing our testbed was scaling our system to handle 200,000-500,000 queries per second across 20 different backend systems.

For example, our initial system configuration used a sharded distributed caching layer. We moved away from this design because the large batch size within some requests (up to 300 queries) meant that every cache had to be accessed [20, 77]. Our current testbed matches the design used in the OneRF production system in that each application server only queries its local cache.

Another challenge we compensate for is the delay of reallocating cache space in off-the-shelf memcached instances. Memcached's reallocation API works at the granularity of 1MB pages. To reallocate 1% of the cache space (Section 3), up to 327 memcached pages need to be reallocated. To reallocate a page, memcached must acquire several locks, in order to safely evict page contents. High load in our experiments leads to memcached-internal lock contention, which delays reallocation steps. Typically (95% of the time), reallocations take no longer than 5 seconds, which is why $\Delta = 5$ seconds (in Section 3). To tolerate atypical reallocations that take longer than 5 seconds, the RobinHood controller can defer cache allocations to future iterations of the RobinHood algorithm.

Finally, we carefully designed our testbed for reproducible performance results. For example, to deal with complex state throughout the deployment (e.g., in various backends), we wipe all state between repeated experiments, at the cost of a longer warmup period.

4.3 Generating experimental data

Microsoft shared with us detailed statistics of production traffic in the OneRF system for several days in 2017 and 2018 (see Section 2). We base our evaluation on the

[4]The servers are provisioned with 2.4 GHz Intel E5-2673v3 with eight cores, 64GB memory, 400GB SSDs with up to 24000 IOPS, and 8Gbit/s network bandwidth.

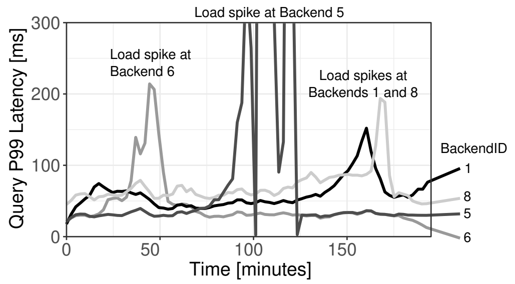


Figure 7: P99 latency of backend queries in our experiment. The four latency spikes emulate the latency spikes in the OneRF production systems (see Figure 2).

2018 dataset. The dataset describes queries to more than 40 distinct backend systems.

In our testbed, we replicate the 20 most queried backend systems, which make up more than 99.95% of all queries. Our backends contain objects sampled from the OneRF object size distribution. Across all backends, object sizes range between a few bytes to a few hundred KB, with a mean of 23 KB. In addition, our backends approximately match the design of the corresponding OneRF backend.

Our request traffic replicates key features of production traffic, such as an abundance of several hundreds of different request types, each with their own request structures (e.g., batch size, fanout, etc). We sample from the production request type distribution and create four-hour-long traces with over 50 million requests and 2.5 billion queries. We verified that our traces preserve statistical correlations and locality characteristics from the production request stream. We also verified that we accurately reproduce the highly varying cacheability of different backend types. For example, the hit ratios of the four backends with latency spikes (Figure 2) range between 81-92% (backend 1), 51-63% (backend 5), 37-44% (backend 6), and 96-98% (backend 8) in our experiments. The lowest hit ratio across all backends is 10% and the highest is 98%, which means that the P99 tail latency is always composed of cache misses (this matches our observations from the production system). None of the working sets fit into the application server's cache, preventing trivial scenarios as mentioned in the literature [26].

5 Evaluation

Our empirical evaluation of RobinHood focuses on five key questions. Throughout this section, our goal is to meet a P99 request latency Service Level Objective (SLO) of 150ms, which is a typical goal for user-facing web applications [26, 27, 34, 50, 56, 59, 90, 91]. Every 5s, we measure the P99 over the previous 60s. We define the SLO violation percentage to be the fraction of observations where the P99 does not meet the SLO. We compare

Name	Optimization goal	Dy-namic	Latency-aware	Request structure
RobinHood	Minimize *request* P99	yes	yes	yes
OneRFpolicy	Minimize miss ratio	no	no	no
TAO$_{++}$ [20]	Minimize *request* P99	no	no	no
Cliffhgr$_{++}$ [25]	Minimize miss ratio	yes	no	no
FAIR$_{++}$ [19,65]	Equalize miss ratios	yes	no	no
LAMA$_{++}$ [18,40]	Equalize *query* P99	yes	yes	no

Table 2: The six caching systems evaluated in our experiments. RobinHood is the only dynamic caching system that seeks to minimize the tail *request* latency and the first caching system that utilizes request structure rather than just queries.

RobinHood to five state-of-the-art caching systems, defined in Section 5.1, and answer the following questions:

Section 5.2: How much does RobinHood improve SLO violations for OneRF's workload? Quick answer: RobinHood brings SLO violations down to 0.3%, compared to 30% SLO violations under the next best policy.

Section 5.3: How much variability can RobinHood handle? Quick answer: for quickly increasing backend load imbalances, RobinHood maintains SLO violations below 1.5%, compared to 38% SLO violations under the next best policy.

Section 5.4: How robust is RobinHood to simultaneous latency spikes? Quick answer: RobinHood maintains less than 5% SLO violations, while other policies do significantly worse.

Section 5.5: How much space does RobinHood save? Quick answer: The best clairvoyant static allocation requires 73% more cache space in order to provide each backend with its maximum allocation under RobinHood.

Section 5.6: What is the overhead of running RobinHood? Quick answer: RobinHood introduces negligible overhead on network, CPU, and memory usage.

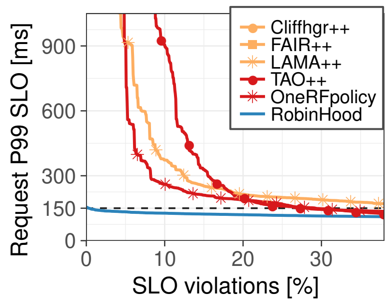
5.1 Competing caching systems

We compare RobinHood to two production systems and three research caching systems listed in Table 2.

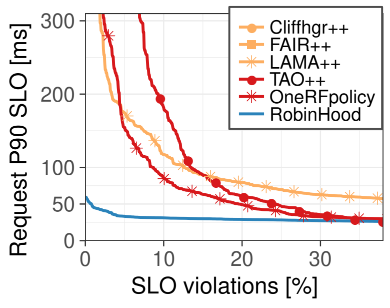
The two production systems do not currently dynamically adjust the caches. OneRFpolicy uses a single shared cache, which matches the configuration used in the OneRF production system. TAO$_{++}$ uses static allocations. As manually deriving the optimal allocation is an open problem [20], we actually use RobinHood to find a good allocation for the first 20% of the experiment in Section 5.2. TAO$_{++}$ then keeps this allocation fixed throughout the experiment. This is an optimistic version of TAO (thus the name TAO$_{++}$) as finding its allocation would have been infeasible without RobinHood.[5].

We evaluate three research systems, Cliffhanger [25], FAIR [19,65,85], and LAMA [40] (which is conceptually

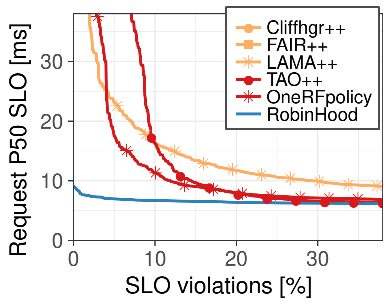
[5]For example , we have also experimented with brute-force searches, but the combinatorial search space for 20 partitions is too large. We did not find a better allocation over the course of 48 hours.



(a) P99 SLOs



(b) P90 SLOs



(c) P50 SLOs

Figure 8: Request SLO as a function of SLO violations for (a) P99 SLOs, (b) P90 SLOs, (c) P50 SLOs. For a given violation percentage the plot shows what SLO would have been violated with that frequency. A lower value indicates a system is able to meet lower latency SLOs with fewer SLO violations. RobinHood is the only system that is robust against latency spikes on backends and violates a 150ms P99 SLO only 0.3% of the time (dashed horizontal line in (a)). FAIR$_{++}$ and Cliffhgr$_{++}$ are not shown as their SLO violations are too high to be visible.

similar to [18]). All three systems dynamically adjust the cache, but required major revisions before we could compare against them. All three research systems are only designed to work on a single cache. Two of the systems, Cliffhanger and FAIR, are not aware of multiple backends, which is typical for application-layer caching systems. They do not incorporate request latency or even query latency, as their goal is to maximize the overall cache hit ratio and the fairness between users, respectively. We adapt Cliffhanger and FAIR to work across distributed application servers by building a centralized statistics server that aggregates and distributes their measurements. We call their improved versions Cliffhgr$_{++}$ and FAIR$_{++}$. LAMA's goal is to minimize mean query latency, not tail query latency (and it does not consider request latency). To make LAMA competitive, we change the algorithm to use P99 query latency and a centralized statistics server. We call this improved version LAMA$_{++}$.

Our evaluation does not include cost-aware replacement policies for shared caches, such as Greedy-Dual [21] or GD-Wheel [52]. Due to their high complexity, it is challenging to implement them in concurrent caching systems [11]; we are not aware of any production system that implements these policies. Moreover, the OneRF workload does not meet the basic premise of cost-aware caching (Section 2.2.2).

5.2 How much does RobinHood improve SLO violations for OneRF's workload?

To compare RobinHood to the five caching systems above, we examine a scenario that replicates the magnitude and rate with which query latency varies over time in the OneRF production system, as shown in Figure 2. In production systems, this variability is often caused by temporary spikes in the traffic streams of other services which share these backends (Section 2).

Experimental setup. To make experiments reproducible, we emulate latency imbalances by imposing a variable

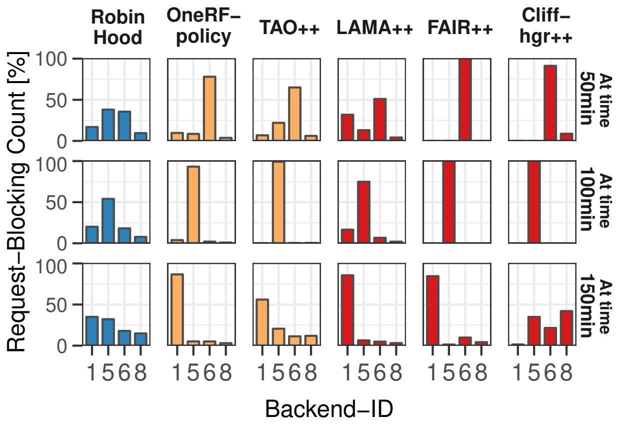


Figure 9: Comparison of how well different caching systems balance the RBC at three times in the experiment from Figure 7. RobinHood is the only system whose RBCs do not significantly exceed 50%.

resource limit on several backends in our testbed. Depending on the backend type (see Section 4), a backend is either IO-bound or CPU-bound. We use Linux control groups to limit the available number of IOPS or the CPU quota on the respective backends. For example, to emulate the latency spike on Backend 6 at 4AM (Figure 2), we limit the number of IOPS that this backend is allowed to perform, which mimics the effect of other traffic streams consuming these IOPS.

We emulate latency variability across the same four backends as shown in Figure 2: backends 1, 5, 6, and 8. Our experiments span four hours each, and we use the first 25% of the experiment time to warm the backends and caches. Figure 7 shows the P99 latency of queries in our experiments under the OneRFpolicy (ignoring an initial warmup period). We verified that the latency spikes are similar in magnitude to those we observe in the OneRF production system (Figure 2).

Experimental results. We compare RobinHood to the

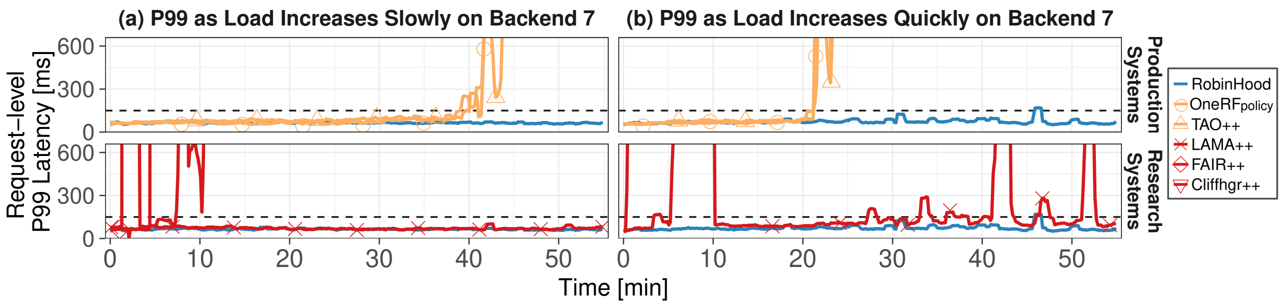


Figure 10: Results from sensitivity experiments where *latency is uncorrelated with query rate*. The load on Backend 7 increases either slowly (left column) or quickly (right column). Even when load increases quickly, RobinHood violates a 150ms P99 SLO less than 1.5% of the time. In contrast, the second best system (LAMA$_{++}$) has 38% SLO violations.

competing systems along several dimensions: the P99 request latency, the rate of SLO violations, and how well they balance the RBC between backends.

Figure 3 shows the P99 request latency for each system over the course of the experiment. Throughout the experiment, RobinHood maintains a P99 below our SLO target of 150ms. Both the production and research systems experience high P99 latencies during various latency spikes, and Cliffhgr$_{++}$ and FAIR$_{++}$ even experience prolonged periods of instability. RobinHood improves the P99 over every competitor by at least 3x during some latency spike.

Figure 8 summarizes the frequency of SLO violations under different SLOs for each caching system in terms of the P99, P90 and P50 request latency. If the goal is to satisfy the P99 SLO 90% of the time, then the graph indicates the strictest latency SLO supported by each system (imagine a vertical line at 10% SLO violations). If the goal is to meet a particular P99 SLO such as 150ms, then the graph indicates the fraction of SLO violations (imagine a horizontal line at 150ms). The figure does not show FAIR$_{++}$ and Cliffhgr$_{++}$ as the percentage of SLO violations is too high to be seen. We find that RobinHood can meet much lower latency SLOs than competitors with almost no SLO violations. For example, RobinHood violates a P99 SLO of 150ms (Figure 8a) less than 0.3% of the time. By contrast, the next best policy, OneRFpolicy, violates this same SLO 30% of the time.

Throughout these experiments, RobinHood targets the P99 request latency (Section 3). We discuss in Section 6 how to generalize RobinHood's optimization goal. However, even though RobinHood focuses on the P99, it still performs well on the P90 and P50. For example, for a P90 SLO of 50ms (Figure 8b), RobinHood leads to less than 1% SLO violations, whereas OneRFpolicy leads to about 20% SLO violations.

Figure 9 shows the RBC (defined in Section 3) for the four backends affected by latency spikes under each caching system at 50 min, 100 min, and 150 min into the experiment, respectively. This figure allows us to quantify how well each system uses the cache to balance RBCs across backend systems. We refer to the *dominant backend* as the backend which accounts for the highest percentage of RBCs. RobinHood achieves the goal of maintaining a fairly even RBC balance between backends—in the worst case, RobinHood allows the dominant backend to account for 54% of the RBC. No other competitor is able to keep the dominating backend below 85% in all cases and even the average RBC of the dominant backend exceeds 70%.

5.3 How much variability can RobinHood handle?

To understand the sensitivity of each caching policy to changes in backend load, we perform a more controlled sensitivity analysis.

Experimental setup. To emulate the scenario that some background work is utilizing the resources of a backend, we limit the resources available to a backend system. In these experiments, we continuously decrease the resource limit on a single backend over a duration of 50 minutes. We separately examine two backends (backend 1 and backend 7) and test two different rates for the resource decrease—the "quick" decrease matches the speed of the fastest latency spikes in the OneRF production system, and the "slow" decrease is about one third of that speed.

Experimental results. Figure 10 shows the P99 request latency under increasing load on backend 7. This experiment benchmarks the typical case where *high latency is uncorrelated with query rate* (Section 2). Figure 10(a) shows that, when load increases slowly, RobinHood never violates a 150ms SLO. In contrast, OneRFpolicy and TAO$_{++}$ are consistently above 150ms after 40min, when the latency imbalance becomes more severe than in Section 5.2. Of the research systems, FAIR$_{++}$ and Cliffhgr$_{++}$ are above 150ms after 10min. LAMA$_{++}$, the only system that is latency aware, violates the SLO 3.3% of the time.

Figure 10(b) shows that, when load increases quickly, RobinHood maintains less than 1.5% SLO violations. All other systems become much worse, e.g., OneRFpolicy and TAO$_{++}$ are above 150ms already after 20min. The second best system, LAMA$_{++}$, violates the SLO more

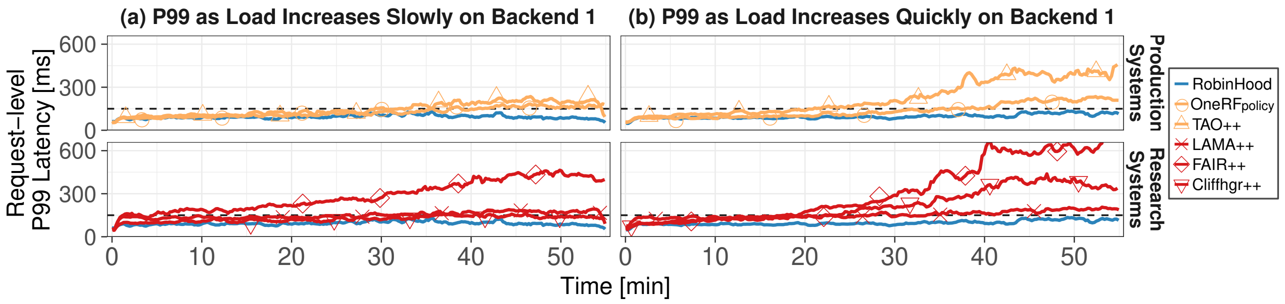


Figure 11: Results from sensitivity experiments where *latency is correlated with query rate*. The load on Backend 1 increases either slowly (left column) or quickly (right column). Even when load increases quickly, RobinHood never violates a 150ms P99 SLO. In contrast, the second best system (OneRFpolicy) has 33% SLO violations.

than 38% of the time, which is 25× more frequent than RobinHood.

Figure 11 shows the P99 request latency under increasing load on backend 1, where high latency is *correlated with query rate*, which is not typical in production systems. Figure 11(a) shows that, when load increases slowly, RobinHood never violates a 150ms SLO. OneRFpolicy and TAO$_{++}$ lead to lower P99s than when latency is uncorrelated, but still violate the 150ms SLO more than 28% of the time. Of the research systems, Cliffhgr$_{++}$ is the best with about 8.2% SLO violations.

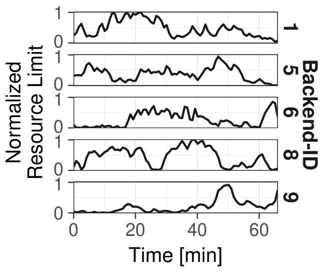
Figure 11(b) shows that, when load increases quickly, RobinHood never violates the SLO. The second best system, OneRFpolicy, violates the SLO more than 33% of the time.

5.4 How robust is RobinHood to simultaneous latency spikes?

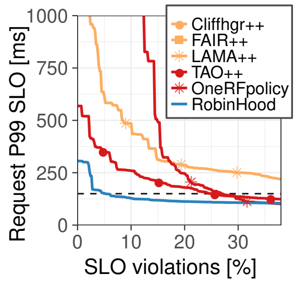
To test the robustness of RobinHood, we allow the backend resource limits to vary randomly and measure RobinHood's ability to handle a wide range of latency imbalance patterns.

Experimental setup. We adjust resource limits over time for the same backends and over the same ranges as those used in the experiments from Section 5.2. However, rather than inducing latency spikes similar to those in the OneRF production system, we now allow resource limits to vary randomly. Hence, each backend will have multiple periods of high and low latency over the course of the experiment. Additionally, multiple backends may experience high latency at the same time. To generate this effect, we randomly increase or decrease each backend's resource limit with 50% probability every 20 seconds.

Experimental results. Figure 12 shows the results of our robustness experiments. Figure 12(a) shows the backend resource limits (normalized to the limits in Section 5.2) over time for each of the backends that were resource limited during the experiment. Note that at several times during the experiment, *multiple backends* were highly limited at the same time, making it more difficult to maintain low request tail latency.



(a) Randomized Resource Limits.



(b) SLO violations.

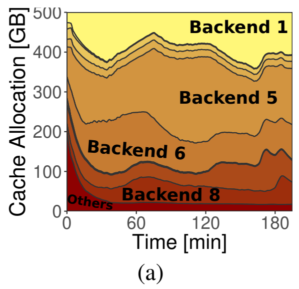
Figure 12: Results from robustness experiments in which backend resource limits vary randomly, see (a). In this challenging scenario, RobinHood still meets 150ms P99 SLO 95% of the time, see (b).

Figure 12(b) shows the rate of SLO violations for each caching system during the robustness experiments. In this challenging scenario, RobinHood violates a 150ms SLO only 5% of the time. The next best policy, TAO$_{++}$, violates the SLO more than 24% of the time. RobinHood also helps during parts of the experiment where all backends are severely resource constrained. Overall, RobinHood's maximum P99 latency does not exceed 306ms whereas the next best policy, TAO$_{++}$, exceeds 610ms.

We observe that the there is no single "second best" caching system: the order of the competitors OneRFpolicy, TAO$_{++}$, and LAMA$_{++}$ is flipped between Figures 12 and 8. In Figure 12(b), TAO$_{++}$ performs well by coincidence and not due to an inherent advantage of static allocations. TAO$_{++}$'s static allocation is optimized for the first part of the experiment shown in Figure 7, where a latency spike occurs on backend 6. Coincidentally, throughout our randomized experiment, backend 6 is also severely resource limited, which significantly boosts TAO$_{++}$'s performance.

5.5 How much space does RobinHood save?

Figure 13 shows RobinHood's allocation per backend in the experiments from Sections 5.2 and 5.4. To get an



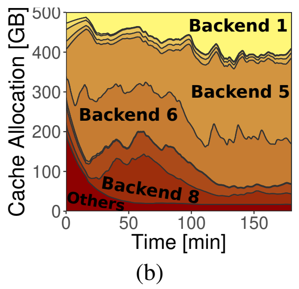


Figure 13: RobinHood's overall cache allocation during the experiments from Sections 5.2 and 5.4.

estimate of how much space RobinHood saves over other caching systems, we consider what static allocation would be required by TAO$_{++}$ in order to provide each backend with its maximum allocation under RobinHood. This significantly underestimates RobinHood's advantage, since it assumes the existence of an oracle that knows RobinHood's allocations ahead of time. Even given advance knowledge of these allocations, TAO$_{++}$ would need 73% more cache space than RobinHood.

5.6 What is the overhead of running RobinHood?

We consider three potential sources of overhead.

Network overhead. In our implementation of RobinHood, application servers send request statistics to an RBC server (Section 3) once every second. These updates include the request latency (32-bit integer) and the ID of the request's blocking backend (32-bit integer). Given a request rate of 1000 requests per second per application server, this amounts to less than 8 KB/s.

CPU and memory overhead. RobinHood adds a lightweight controller to each application server (Section 3). Throughout all experiments, the controller's CPU utilization overhead was too small to be measured. The memory overhead including RobinHood's controller is less than 25 KB. However, we measured bursts of memory overhead up to several MBs. This is due to memcached not freeing pages immediately after completing deallocation requests. Future implementations could address these bursts by refining the memcached resizing mechanism.

Query hit latency overhead. In multi-threaded caching systems, such as memcached, downsizing a partition will cause some concurrent cache operations to block (Section 4). We quantify this overhead by measuring the P99 cache hit latency for queries in RobinHood and OneRF for the five backends with the largest change in partition sizes (backends 1, 5, 6, 8, and 9). These measurements are shown in Figure 14. RobinHood increases the P99 cache hit latency for queries by 13% to 28% for the five backends, but does not significantly affect the other backends. Importantly, recall that request latency is different from query latency. The cause of high *request* tail latency is almost solely due to cache misses. Consequently, these hit latencies do not increase the request latency of Robin-

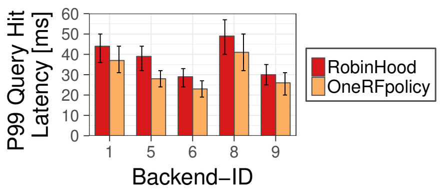


Figure 14: The cache hit latencies under RobinHood and the OneRFpolicy. The small overhead introduced by RobinHood does not affect request tail latency.

Hood even for low percentiles (cf. the P50 request latency in Figure 8c).

Query miss latency overhead. RobinHood can increase the load on cache-rich backends. Across all experiments, the worse-case increase of an individual backend (backend 20, the least queried backend), is 2.55× over OneRF. Among the top 5 backends, RobinHood never increases query latencies by more than 61%. On the other hand, RobinHood improves the P99 query latency by more than 4× for the overloaded backend during the first latency spike in Figure 7. By sacrificing the performance of cache-rich backends, RobinHood frees up cache space to allocate towards cache-poor backends that are contributing to slow request latency. This trade-off significantly improves the request latency both at the P99 as well as at other percentiles (Section 5.2).

6 Discussion

We have seen that RobinHood is capable of meeting a 150ms SLO for the OneRF workload even under challenging conditions where backends simultaneously become overloaded. Many other systems, e.g., at Facebook [20], Google [26], Amazon [27], and Wikipedia [14], use a similar multitier architecture where a request depends on many queries. However, these other systems may have different optimization goals, more complex workloads, or slight variations in system architecture compared to OneRF. In this section, we discuss some of the challenges that may arise when incorporating RobinHood into these other systems.

Non-convex miss curves. Prior work has observed non-convex miss ratio curves (a.k.a. performance cliffs) for some workloads [12, 25, 73, 80]. This topic was also frequently raised in our discussions with other companies. While miss ratio curves in our experiments are largely convex, RobinHood does not fundamentally rely on convexity. Specifically, RobinHood never gets stuck, because it ignores the miss ratio slope. Nevertheless, non-convexities can lead to inefficiency in RobinHood's allocation. If miss ratio curves are highly irregular (step functions), we suggest convexifying miss ratios using existing techniques such as Talus [12] and Cliffhanger [25].

Scaling RobinHood to more backend systems and higher request rates. The RobinHood algorithm scales linearly in the number of backend systems and thus can support hundreds of backends (e.g. services in a microservice architecture). Even at high request rates, RobinHood's overhead is only a few MB/s for up to a million requests per second (independent of the query rate). At a sufficiently high request rate, RobinHood's central RBC server may become the bottleneck. However, at this scale, we expect that it will no longer be necessary to account for every request when calculating the RBC. Sampling some subset of the traffic will still produce a P99 estimate with enough observations to accurately depict the system state. It is also worth noting that typical production systems already have latency measurement systems in place [77] and thus would not require a dedicated RBC server.

Interdependent backends. RobinHood assumes that query latencies are independent across different backends. In some architectures, however, multiple backends share the same underlying storage system [3]. If this shared storage system were the bottleneck, allocating cache space to just one of the backends may be ineffective. RobinHood needs to be aware of such interdependent backends. A future version of RobinHood could fix this problem by grouping interdependent backends into a single unit for cache allocations.

Multiple webservices with shared backends. Optimizing tail latencies across multiple webservices which make use of the same, shared backend systems is challenging. RobinHood can introduce additional challenges. For example, one webservice running RobinHood may increase the load significantly on a shared backend which can negatively affect request latencies in a second webservice. This could arise if the two services see differently structured requests—the shared backend could seem unimportant to one webservice but be critical to another. If both webservices run RobinHood, a shared backend's load might oscillate between low and high as the two RobinHood instances amplify the effect of each other's allocation decisions. A solution to this problem could be to give RobinHood controllers access to the RBC servers of both webservices (effectively running a global RobinHood instance). If this is impossible, additional constraints on RobinHood's allocation decisions could be necessary. For example, we can constrain RobinHood to assign at least as much capacity to the shared backend as it would get in a system without RobinHood.

Distributed caching. Many large webservices rely on a distributed caching layer. While these layers have access to large amounts of cache capacity, working sets typically still do not fit into the cache and the problem of tuning partition sizes remains [20]. RobinHood can accommodate this scenario with solely a configuration change, associating a RobinHood controller with each cache rather than each application server. We have tested RobinHood in this configuration and verified the feasibility of our proposal. However, distributed caching leads to the known problem of cache hotspots under the OneRF workload, regardless of whether or not RobinHood is running (see Section 4.2 and [20, 77]). Addressing this issue is outside the scope of this work, and hence we focus on the cache topology used by OneRF rather than a distributed caching layer.

Scenarios where cache repartitioning is not effective. If the caching layer is severely underprovisioned, or if the workload is highly uncacheable, repartitioning the cache might not be sufficient to reduce P99 request latency. However, we note that RobinHood's key idea—allocating resources to backends which affect P99 request latency—can still be exploited. For instance, if caching is ineffective but backends can be scaled quickly, the RBC metric could be used to drive these scaling decisions in order to reduce request tail latency. Even if backends are not scalable, RBC measurements collected over the course of a day could inform long-term provisioning decisions.

Performance goals beyond the P99. Depending on the nature of the application, system designers may be concerned that a using single optimization metric (e.g., P99) could lead to worse performance with respect to other metrics (e.g., the average request latency). However, RobinHood explicitly optimizes whatever optimization metric is used to calculate the RBC. Hence, it is possible to use other percentiles or even multiple percentiles to calculate the RBC by choosing the set S accordingly (see Section 3 for a definition of S). Conceptually, RobinHood is modular with regard to both the resources it allocates and with regard to the metric that is used to drive these allocations.

7 Related Work

A widely held opinion is that "caching layers ... do not directly address tail latency, aside from configurations where it is guaranteed that the entire working set of an application can reside in a cache" [26]. RobinHood is the first work that shows that *caches can directly address tail latency even if working sets are much larger than the cache size*. Thus, RobinHood stands at the intersection of two bodies of work: caching and tail latency reduction.

Caching related work. Caching is a heavily studied area of research ranging from theory to practice [10]. For the most part, the caching literature has primarily focused on improving hit ratios (e.g., [2, 8, 13, 15–17, 21, 22, 42, 57, 87]). Prior work has also investigated strategies for dynamically partitioning a cache to maximize overall hit ratio (e.g., [1, 24, 25, 40, 60, 75]) or to provide a weighted or fair hit ratio to multiple workloads (e.g., [19,46,65,85]). Importantly, while hit ratio is a good proxy for average latency, it does not capture the effect of tail latency, which is dominated by the backend system performance.

Another group of caching policies incorporates "miss cost" (such as retrieval latency) into eviction decisions [18, 21, 33, 52, 64, 70, 81, 86]. As discussed in Section 2.2.2, the OneRF workload does not meet the premise of cost-aware caching. Specifically, all these systems assume that the retrieval latency is correlated (fixed) per object. At OneRF, latency is highly variable over time and not correlated with specific objects.

The most relevant systems are LAMA [40] and Hyperbolic [18]. LAMA partitions the cache by backend and seeks to balance the average latency across backends. Hyperbolic does not support partitions, but allows estimating a metric across a group of related queries such as all queries going to the same backend. This enables Hyperbolic to work similarly to LAMA. Both LAMA and Hyperbolic are represented optimistically by LAMA$_{++}$ in our evaluation. Unfortunately, LAMA$_{++}$ leads to high P99 request latency because the latency of individual queries is typically not a good indicator for the overall request tail latency (see Section 2). Unlike LAMA or Hyperbolic, RobinHood directly incorporates the request structure in its caching decisions.

Another branch of works seeks to improve the caching system itself, e.g., the throughput [31,66], the latency of cache hits [67], cache-internal load balancing [32,43], and cache architecture [31,55,72]. However, these works are primarily concerned with the performance of cache hits rather than cache misses, which dictate the overall request tail latency.

Tail latency related work. Reducing tail latency and mitigating stragglers is an important research area that has received much attention in the past decade. Existing techniques can be subdivided into the following categories: redundant requests, scheduling techniques, auto-scaling and capacity provisioning techniques, and approximate computing. Our work serves to introduce a fifth category: *using the cache to reduce tail latency.*

A common approach to mitigating straggler effects is to send redundant requests and use the first completed request [5–7,45,69,78,79,82,84,88]. When requests cannot be replicated, prior work has proposed several scheduling techniques, e.g., prioritization strategies [36,89,92], load balancing techniques [48,53,83], and systems that manage queueing effects [4,28,29,54,62,68,74]. These are useful techniques for cutting long tail latencies, but fundamentally, they still have to send requests to backend systems, whereas our new caching approach eliminates a fraction of traffic to backend systems entirely.

While these first two approaches consider systems with static resource constraints, other works have considered adjusting the overall compute capacity to improve tail latency. These techniques include managing the compute capacity through auto-scaling and capacity provisioning for clusters [34,45,47,58,90,91], and adjusting the power and/or compute (e.g., number of cores) allocated to performing the computation [37,38,49,76]. Alternatively, there is a branch of tail latency reduction work known as approximate computing, which considers reducing the computational requirements by utilizing lower quality results [6,23,45,51]. Importantly, these are all orthogonal approaches for reducing tail latency, and our work is proposing a new type of technique that can be layered on top of these existing techniques.

Why RobinHood is different. RobinHood is unique in several ways. First, it is the only system to utilize the cache for reducing overall request tail latency. Second, RobinHood is the only caching system that takes request structure into account. Third, by operating at the caching layer, RobinHood is uniquely situated to influence many diverse backend systems without requiring any modifications to the backend systems.

8 Conclusion

This paper addresses two problems facing web service providers who seek to maintain low request tail latency. The first problem is to determine the best allocation of resources in multitier systems which serve structured requests. To deal with structured requests, RobinHood introduces the concept of the request blocking count (RBC) for each backend, identifying which backends require additional resources. The second problem is to address latency imbalance across stateful backend systems which cannot be scaled directly to make use of the additional resources. RobinHood leverages the existing caching layer present in multitiered systems, differentially allocating cache space to the various backends in lieu of being able to scale them directly.

Our evaluation shows that RobinHood can reduce SLO violations from 30% to 0.3% for highly variable workloads such as OneRF. RobinHood is also lightweight, scalable, and can be deployed on top of an off-the-shelf software stack. The RobinHood caching system demonstrates how to effectively identify the root cause of P99 request latency in the presence of structured requests. Furthermore, RobinHood shows that, contrary to popular belief, a properly designed caching layer *can* be used to reduce higher percentiles of request latency.

Acknowledgments

We thank Jen Guriel, Bhavesh Thaker, Omprakash Maity, and everyone on the OneRF team at Microsoft. We also thank the anonymous reviewers, and our shepherd, Frans Kaashoek, for their feedback. This paper was supported by NSF-CSR-180341, NSF-XPS-1629444, NSF-CMMI-1538204, and a Faculty Award from Microsoft.

References

[1] C. L. Abad, A. G. Abad, and L. E. Lucio. Dynamic memory partitioning for cloud caches with heteroge-

neous backends. In *ACM ICPE*, pages 87–90, New York, NY, USA, 2017. ACM.

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

[3] C. Albrecht, A. Merchant, M. Stokely, M. Waliji, F. Labelle, N. Coehlo, X. Shi, and E. Schrock. Janus: Optimal flash provisioning for cloud storage workloads. In *USENIX ATC*, pages 91–102, 2013.

[4] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda. Less is more: trading a little bandwidth for ultra-low latency in the data center. In *USENIX NSDI*, pages 19–19, 2012.

[5] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Effective straggler mitigation: Attack of the clones. In *USENIX NSDI*, pages 185–198, Berkeley, CA, USA, 2013. USENIX Association.

[6] G. Ananthanarayanan, M. C.-C. Hung, X. Ren, I. Stoica, A. Wierman, and M. Yu. GRASS: Trimming stragglers in approximation analytics. In *USENIX NSDI*, pages 289–302, Seattle, WA, 2014. USENIX Association.

[7] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in map-reduce clusters using mantri. In *USENIX OSDI*, pages 265–278, Berkeley, CA, USA, 2010. USENIX Association.

[8] M. Arlitt, L. Cherkasova, J. Dilley, R. Friedrich, and T. Jin. Evaluating content management techniques for web proxy caches. *Performance Evaluation Review*, 27(4):3–11, 2000.

[9] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS*, pages 53–64, 2012.

[10] A. Balamash and M. Krunz. An overview of web caching replacement algorithms. *IEEE Communications Surveys & Tutorials*, 6(2):44–56, 2004.

[11] N. Beckmann, H. Chen, and A. Cidon. LHD: Improving cache hit rate by maximizing hit density. In *USENIX NSDI*, pages 389–403, 2018.

[12] N. Beckmann and D. Sanchez. Talus: A simple way to remove cliffs in cache performance. In *IEEE HPCA*, pages 64–75, 2015.

[13] D. S. Berger, N. Beckmann, and M. Harchol-Balter. Practical bounds on optimal caching with variable object sizes. *POMACS*, 2(2):32, 2018.

[14] D. S. Berger, B. Berg, T. Zhu, and M. Harchol-Balter. The case of dynamic cache partitioning for tail latency, March 2017. Poster presented at USENIX NSDI.

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

[16] D. S. Berger, S. Henningsen, F. Ciucu, and J. B. Schmitt. Maximizing cache hit ratios by variance reduction. *ACM SIGMETRICS Perform. Eval. Rev.*, 43(2):57–59, Sept. 2015.

[17] D. S. Berger, R. K. Sitaraman, and M. Harchol-Balter. AdaptSize: Orchestrating the hot object memory cache in a content delivery network. In *USENIX NSDI*, pages 483–498, Berkeley, CA, USA, 2017. USENIX Association.

[18] A. Blankstein, S. Sen, and M. J. Freedman. Hyperbolic caching: Flexible caching for web applications. In *USENIX ATC*, pages 499–511, 2017.

[19] J. Brock, C. Ye, C. Ding, Y. Li, X. Wang, and Y. Luo. Optimal cache partition-sharing. In *IEEE ICPP*, pages 749–758, 2015.

[20] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. C. Li, et al. Tao: Facebook's distributed data store for the social graph. In *USENIX ATC*, pages 49–60, 2013.

[21] P. Cao and S. Irani. Cost-aware WWW proxy caching algorithms. In *USENIX Symposium on Internet Technologies and Systems*, 1997.

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

[23] M. Chow, K. Veeraraghavan, M. Cafarella, and J. Flinn. Dqbarge: Improving data-quality tradeoffs in large-scale internet services. In *USENIX OSDI*, pages 771–786, Savannah, GA, 2016. USENIX Association.

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

[25] A. Cidon, A. Eisenman, M. Alizadeh, and S. Katti. Cliffhanger: Scaling performance cliffs in web memory caches. In *USENIX NSDI*, 2016.

[26] J. Dean and L. A. Barroso. The tail at scale. *CACM*, 56(2):74–80, 2013.

[27] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *ACM SOSP*, volume 41, pages 205–220, 2007.

[28] P. Delgado, D. Didona, F. Dinu, and W. Zwaenepoel. Job-aware scheduling in eagle: Divide and stick to your probes. In *ACM SoCC*, pages 497–509, New York, NY, USA, 2016. ACM.

[29] P. Delgado, F. Dinu, A.-M. Kermarrec, and W. Zwaenepoel. Hawk: Hybrid datacenter scheduling. In *USENIX ATC*, pages 499–510, Berkeley, CA, USA, 2015. USENIX Association.

[30] C. Delimitrou and C. Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. In *ACM ASPLOS*, pages 127–144, 2014.

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

[32] B. Fan, H. Lim, D. G. Andersen, and M. Kaminsky. Small cache, big effect: Provable load balancing for randomly partitioned cluster services. In *ACM SoCC*, pages 23:1–23:12, New York, NY, USA, 2011. ACM.

[33] B. C. Forney, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Storage-aware caching: Revisiting caching for heterogeneous storage systems. In *USENIX FAST*, pages 5–5, Berkeley, CA, USA, 2002. USENIX Association.

[34] A. Gandhi, M. Harchol-Balter, R. Raghunathan, and M. A. Kozuch. Autoscale: Dynamic, robust capacity management for multi-tier data centers. *ACM TOCS*, 30(4):14, 2012.

[35] P. Garraghan, X. Ouyang, R. Yang, D. McKee, and J. Xu. Straggler root-cause and impact analysis for massive-scale virtualized cloud datacenters. *IEEE Transactions on Services Computing*, 2016.

[36] M. P. Grosvenor, M. Schwarzkopf, I. Gog, R. N. Watson, A. W. Moore, S. Hand, and J. Crowcroft. Queues don't matter when you can jump them! In *USENIX NSDI*, pages 9–21, 2015.

[37] M. E. Haque, Y. h. Eom, Y. He, S. Elnikety, R. Bianchini, and K. S. McKinley. Few-to-many: Incremental parallelism for reducing tail latency in interactive services. In *ASPLOS*, pages 161–175, New York, NY, USA, 2015. ACM.

[38] M. E. Haque, Y. He, S. Elnikety, T. D. Nguyen, R. Bianchini, and K. S. McKinley. Exploiting heterogeneity for tail latency and energy efficiency. In *IEEE/ACM MICRO*, pages 625–638, New York, NY, USA, 2017. ACM.

[39] M. Harchol-Balter. *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*. Cambridge University Press, 2013.

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

[41] Q. Huang, K. Birman, R. van Renesse, W. Lloyd, S. Kumar, and H. C. Li. An analysis of Facebook photo caching. In *SOSP*, 2013.

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

[43] J. Hwang and T. Wood. Adaptive performance-aware distributed memory caching. In *Proceedings of the International Conference on Autonomic Computing*, pages 33–43, San Jose, CA, 2013. USENIX.

[44] V. Jalaparti, P. Bodik, S. Kandula, I. Menache, M. Rybalkin, and C. Yan. Speeding up distributed request-response workflows. In *ACM SIGCOMM*, pages 219–230, 2013.

[45] V. Jalaparti, P. Bodik, S. Kandula, I. Menache, M. Rybalkin, and C. Yan. Speeding up distributed request-response workflows. In *ACM SIGCOMM*, pages 219–230, New York, NY, USA, 2013. ACM.

[46] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely Jr, and J. Emer. Adaptive insertion policies for managing shared caches. In *ACM PACT*, pages 208–219, 2008.

[47] K. Jang, J. Sherry, H. Ballani, and T. Moncaster. Silo: Predictable message latency in the cloud. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 435–448, New York, NY, USA, 2015. ACM.

[48] S. A. Javadi and A. Gandhi. DIAL: reducing tail latencies for cloud applications via dynamic

interference-aware load balancing. In *International Conference on Autonomic Computing*, pages 135–144, 2017.

[49] H. Kasture, D. B. Bartolini, N. Beckmann, and D. Sanchez. Rubik: Fast analytical power management for latency-critical systems. In *IEEE/ACM MICRO*, pages 598–610, Dec 2015.

[50] A. Krioukov, P. Mohan, S. Alspaugh, L. Keys, D. Culler, and R. Katz. Napsac: Design and implementation of a power-proportional web cluster. *ACM SIGCOMM*, 41(1):102–108, 2011.

[51] G. Kumar, G. Ananthanarayanan, S. Ratnasamy, and I. Stoica. Hold 'em or fold 'em?: Aggregation queries under performance variations. In *ACM EUROSYS*, pages 7:1–7:14, 2016.

[52] C. Li and A. L. Cox. GD-Wheel: A cost-aware replacement policy for key-value stores. In *ACM EUROSYS*, pages 5:1–5:15, 2015.

[53] J. Li, K. Agrawal, S. Elnikety, Y. He, I.-T. A. Lee, C. Lu, and K. S. McKinley. Work stealing for interactive services to meet target latency. In *ACM PPoPP*, pages 14:1–14:13, New York, NY, USA, 2016.

[54] J. Li, N. K. Sharma, D. R. K. Ports, and S. D. Gribble. Tales of the tail: Hardware, os, and application-level sources of tail latency. In *ACM SoCC*, pages 9:1–9:14, New York, NY, USA, 2014. ACM.

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

[56] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis. Towards energy proportionality for large-scale latency-critical workloads. In *ACM ISCA*, volume 42, pages 301–312, 2014.

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

[58] A. H. Mahmud, Y. He, and S. Ren. Bats: Budget-constrained autoscaling for cloud performance optimization. In *International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 232–241, Oct 2015.

[59] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, and T. F. Wenisch. Power management of online data-intensive services. In *ACM ISCA*, volume 39, pages 319–330, 2011.

[60] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, et al. Scaling memcache at Facebook. In *USENIX NSDI*, pages 385–398, 2013.

[61] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling memcache at facebook. In *NSDI*, 2013.

[62] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: Distributed, low latency scheduling. In *ACM SOSP*, pages 69–84, New York, NY, USA, 2013. ACM.

[63] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal. Fastpass: A centralized zero-queue datacenter network. In *ACM SIGCOMM*, pages 307–318, 2014.

[64] R. Prabhakar, S. Srikantaiah, C. Patrick, and M. Kandemir. Dynamic storage cache allocation in multi-server architectures. In *Conference on High Performance Computing Networking, Storage and Analysis*, pages 8:1–8:12, New York, NY, USA, 2009. ACM.

[65] Q. Pu, H. Li, M. Zaharia, A. Ghodsi, and I. Stoica. Fairride: Near-optimal, fair cache sharing. In *USENIX NSDI*, pages 393–406, 2016.

[66] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *IEEE/ACM MICRO*, pages 423–432, 2006.

[67] K. Rashmi, M. Chowdhury, J. Kosaian, I. Stoica, and K. Ramchandran. Ec-cache: Load-balanced, low-latency cluster caching with online erasure coding. In *USENIX OSDI*, pages 401–417, 2016.

[68] W. Reda, M. Canini, L. Suresh, D. Kostić, and S. Braithwaite. Rein: Taming tail latency in key-value stores via multiget scheduling. In *EuroSys*, pages 95–110, New York, NY, USA, 2017. ACM.

[69] X. Ren, G. Ananthanarayanan, A. Wierman, and M. Yu. Hopper: Decentralized speculation-aware cluster scheduling at scale. In *ACM SIGCOMM*, pages 379–392, New York, NY, USA, 2015. ACM.

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

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

[72] I. Stefanovici, E. Thereska, G. O'Shea, B. Schroeder, H. Ballani, T. Karagiannis, A. Rowstron, and T. Talpey. Software-defined caching: Managing caches in multi-tenant data centers. In *ACM SoCC*, pages 174–181, New York, NY, USA, 2015. ACM.

[73] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. *The Journal of Supercomputing*, 28(1):7–26, 2004.

[74] L. Suresh, M. Canini, S. Schmid, and A. Feldmann. C3: Cutting tail latency in cloud data stores via adaptive replica selection. In *USENIX NSDI*, pages 513–527, Oakland, CA, 2015. USENIX Association.

[75] J. Tan, G. Quan, K. Ji, and N. Shroff. On resource pooling and separation for lru caching. *Proc. ACM Meas. Anal. Comput. Syst.*, 2(1):5:1–5:31, Apr. 2018.

[76] B. Vamanan, H. B. Sohail, J. Hasan, and T. N. Vijaykumar. Timetrader: Exploiting latency tail to save datacenter energy for online search. In *ACM/IEEE MICRO*, pages 585–597, New York, NY, USA, 2015. ACM.

[77] K. Veeraraghavan, J. Meza, D. Chou, W. Kim, S. Margulis, S. Michelson, R. Nishtala, D. Obenshain, D. Perelman, and Y. J. Song. Kraken: leveraging live traffic tests to identify and resolve resource utilization bottlenecks in large scale web services. In *USENIX OSDI*, pages 635–650, 2016.

[78] A. Vulimiri, P. B. Godfrey, R. Mittal, J. Sherry, S. Ratnasamy, and S. Shenker. Low latency via redundancy. In *ACM CoNEXT*, pages 283–294, New York, NY, USA, 2013. ACM.

[79] A. Vulimiri, O. Michel, P. B. Godfrey, and S. Shenker. More is less: Reducing latency via redundancy. In *ACM HotNets*, pages 13–18, New York, NY, USA, 2012. ACM.

[80] C. Waldspurger, T. Saemundsson, I. Ahmad, and N. Park. Cache modeling and optimization using miniature simulations. In *USENIX ATC*, pages 487–498, 2017.

[81] R. P. Wooster and M. Abrams. Proxy caching that estimates page load delays. *Computer Networks and ISDN Systems*, 29(8):977–986, 1997.

[82] Z. Wu, C. Yu, and H. V. Madhyastha. Costlo: Cost-effective redundancy for lower latency variance on cloud storage services. In *USENIX NSDI*, pages 543–557, Oakland, CA, 2015. USENIX Association.

[83] Y. Xu, Z. Musgrave, B. Noble, and M. Bailey. Bobtail: Avoiding long tails in the cloud. In *USENIX NSDI*, pages 329–341, Lombard, IL, 2013. USENIX.

[84] N. J. Yadwadkar, G. Ananthanarayanan, and R. Katz. Wrangler: Predictable and faster jobs using fewer resources. In *ACM SoCC*, pages 26:1–26:14, New York, NY, USA, 2014. ACM.

[85] C. Ye, J. Brock, C. Ding, and H. Jin. Rochester elastic cache utility (recu): Unequal cache sharing is good economics. *International Journal of Parallel Programming*, 45(1):30–44, 2017.

[86] N. E. Young. On-line file caching. In *ACM SODA*, pages 82–86, Philadelphia, PA, USA, 1998. Society for Industrial and Applied Mathematics.

[87] Y. Yu, W. Wang, J. Zhang, and K. B. Letaief. LRC: dependency-aware cache management for data analytics clusters. *CoRR*, abs/1703.08280, 2017.

[88] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *USENIX OSDI*, pages 29–42, Berkeley, CA, USA, 2008. USENIX Association.

[89] H. Zhu and M. Erez. Dirigent: Enforcing qos for latency-critical tasks on shared multicore systems. *SIGPLAN Not.*, 51(4):33–47, Mar. 2016.

[90] T. Zhu, D. S. Berger, and M. Harchol-Balter. SNC-Meister: Admitting more tenants with tail latency SLOs. In *ACM SoCC*, pages 374–387, 2016.

[91] T. Zhu, M. A. Kozuch, and M. Harchol-Balter. Workloadcompactor: Reducing datacenter cost while providing tail latency slo guarantees. In *ACM SoCC*, pages 598–610, New York, NY, USA, 2017. ACM.

[92] T. Zhu, A. Tumanov, M. A. Kozuch, M. Harchol-Balter, and G. R. Ganger. Prioritymeister: Tail latency qos for shared networked storage. In *ACM SoCC*, pages 29:1–29:14, New York, NY, USA, 2014. ACM.