Design a Cache to Maximize Page-Level Cache Hit Ratios

Ruogu Du, Daniel Berger
Carnegie Mellon University

Abstract

We identify a new problem from real life scenario which is different from the one that traditional caching policies aim at. For the approaches trying to solve this problem, the page-level hit ratio (PHR) instead of the object-level hit ratio (OHR) is the metric to be optimized. We propose two new algorithms, SPO and SRPO, using page-level information in priority evaluation and admission policy. We implement the new algorithms and traditional algorithms like LRU, LFU, and a more recent algorithm Hyperbolic in a sampling-based manner. We also conduct experiments on the system for hit ratio and throughput evaluation. Experiment results show the algorithms outperforms LRU, LFU and Hyperbolic in PHR. We also discover SRPO, as well as LRU and LFU, is a sample-robust algorithm, which means its performance doesn't vary significantly when the sample number changes. We also find a larger sample number reduces the scalability of our system.

1 Introduction

When users access Xbox.com, many individual objects from multiples backends will be requested until the complete pages can be shown. In production, the requests consisting of multiple objects are served by Application Servers, which collect objects from their corresponding backends. The Applications Servers are using local caches to reduce redundant collection from backends. Given this highly fan-out architecture, the latency of a single requests depends on the highest latency among the objects, which means a single miss among all the objects in a request will significantly slows the latency. We want to pursue a higher ratio of requests that are hit entirely in cache, defined as the Page-level-hit-ratio (PHR). This is in contrast to the Object-level-hit-ratio (OHR), which only considers individual objects.

The reason why existing works cannot solve our problem is that they are all based on the assumption that object accesses are independent of other objects. However, this assumption no longer holds in our problem. Previous analysis [12] shows the objects contribution to page-level hit ratio isnt proportional to its popularity. As a result, we would like to design the cache in a way that leverages both the popularity of the objects and the relationship between the objects and requests.

The goal of this project is to prove the feasibility of caches that optimize the page-level hit ratio. To achieve this goal, we would like to build a storage system serving requests with multiple objects, with a cache design that considers the relationship between objects and requests to improve the page-level hit ratio.

2 Related Works

To devise our approach to be optimized for page-level hit ratio, there are many ideas that can be borrowed from existing research. Works on cache design continues to emerge to achieve better efficiency, lower time complexity, and lower synchronization costs. A well-known and widely-used design is Least Recently Used (LRU). When eviction happens, the LRU algorithm selects the page that has not been accessed for the longest time. This algorithm was integrated with many commercial systems from CPUs to CDNs. Despite the popularity, researchers have identified disadvantages in LRU and proposed their variants to improve the algorithm. In the meantime, more and more features start being considered in the replacement algorithms.

Several prior works aim at providing new metrics for eviction and admission policies of the cache. Recency, frequency, size, cost, etc. have been gradually taken in to account.

ONeil et al. [19] argue the LRU algorithm could fail to identify pages that are frequently accessed in some specific patterns. Thus, the evictions solely based on the last access could perform poorly. On the ground of their analysis, they proposed LRU-K, which considers the last K accesses of a page. They argue even this algorithm was designed similar to Least Frequently Used (LFU), but they are different because LRU-K can age older access to keep the most recent access patterns.

Abrams et al. [2] studied LRU-MIN, which is a variant that tries to minimize the number of documents replaced by initially considering larger pages for eviction. They also studied LRU-THOLD, which avoids the entrance of pages that are too large for the cache to cause massive eviction. They conclude that when evictions happen LRU cannot outperform other algorithms since it never considers the sizes of pages. They argue that in LRU multiple evictions of small pages still may not be able to make room for a large page but could reduce hit ratio significantly. They think this scenario could be prevented by the eviction policy of LRU-MIN or the admission policy of LRU-THOLD.

Wooster et al. [24] proposed Hybrid, which is aimed at reducing access latency. It considers the sizes of documents and the bandwidth to retrieve them. It would prone to evict small objects with large bandwidth to reduce the latency of a cache miss.

Cao et al. [7] argue that LRUs disadvantages are not considering files sizes and latency. They introduced GreedyDual-Size algorithm which considers size and latency in addition to locality. It links every page in the cache with a value, which is initially set as the cost/size to fetch the page and will be restored after each access. The value of each page will be deducted when the page with the smallest value is evicted. This algorithm is likely to evict large pages and pages that are hard to retrieve.

LFU-DA [3] is a frequency-based algorithm with the similar aging mechanism like GreedyDual-Size. It computes the priority of each object by mutiplying the cost factor and frequency, plus the aging factor L.

Lee et al. [16] argued that there could be a spectrum of algorithms between Least Recently Used (LRU) and Least Frequently Used (LFU). They proposed LRFU, which leverages the complete history of accesses and considers recency as well as frequency. The algorithm associates a value called Combined Recency and Frequency (CRF) with each block. The value is computed via the time elapsed since each access. The weight between recency and frequency can be reflected by definition of the function for CRF calculation. As a result, this algorithm defines a spectrum between LRU and LFU.

Rizzo et al. [21] sought to introduce statistical parameters into cache design. They proposed Lowest relative Value (LRV), which includes an objective function based on a cost/benefit model. The values computed in the model are based on statistical parameters collected by the server. The model defines benefit as the amount of space freed from an evicted file, and the cost as the fetching cost based on connections, time, or traffic.

Starobinski et al. [22] proposed LRU-S. They claimed the algorithm had constant time complexity and efficiency close to the optimal off-line baseline. In this algorithm, a retrieval cost is defined for each object. The goal of the algorithm is to keep the documents with the largest cost times hit frequency in the cache. The algorithm performs in a probabilistic way. In this paper, there is the proof showing the algorithm should converge to the offline optimal baseline.

Jiang et al. [11] argued that algorithms like LRU-K requires additional history information, which increases the complexity and may not be able to stably improve performance. They proposed Low Interface Recency Set (LIRS), which uses Inter-Reference Recency (IRR) as the history information. IRR is the number of other blocks accessed between two consecutive references to the current block. They argued LIRS could catch more opportunities to cache than LRU without significantly increasing algorithm complexity.

Einziger et al. [8] proposed TinyLFU, a frequency-based admission policy. It uses a Counting Bloom Filter (CBF) to approximate an LFU based admission policy. New items and evicted items should be added to TLFU. The winners in TLFU will be eventually added to the cache. A Minimal Increment CBF is an improvement on the original CBF, which uses multiple hash functions and the estimation result should be the minimal count of all hashed values.

Besides the metrics for evictions and admission, there are works that provide novel designs by using different data structures.

Zhou et al. [28] proposed MQ, which is a multi-queue replacement algorithm. The purpose of this algorithm is to solve the problems in the second level caches, in which accesses have different patterns. In MQ multiple queues are maintained for objects that have different lifetimes. They argued the results shows MQ would perform better when the temporal distances between hits are longer.

Meigddo et al. [18] proposed Adaptive Replacement Cache (ARC). They claimed the algorithm is adaptive and could balance the recency and frequency in a continuous manner. The algorithms keep two LRU lists, one of which keeps pages that are seen once since a recent time, while the other keeps the pages that have been seen at least twice. It considers the first list capturing recency and the other capturing frequency. A fixed replacement policy can be applied to both lists while the sizes of the lists are adaptive to workload. They argued the algorithm was scan-resistant because a scan wont flush out all the existing pages in cache.

Huang et al. [10] studied data collected from each layer that serves photos for Facebook. To improve the cache performance, they proposed Quadruply-segmented LRU (S4LRU). The algorithm maintains for queues from level 0 to 3. When an object is added to the queues, it should be initially added to level 0. An item will be alleviated to a higher queue after each cache hit. They presented analysis to show S4LRU could improve hit ratio on edges and original backends.

In addition, there are works focusing on improving the time complexity and latency of the caching systems. Some of them adopted a multiple-queue solution as well. There are also some works using different eviction strategies.

Johnson et al. [15] argued that LRU-K required logarithmic complexity to update the priority queue every time when an access happens. They proposed Two Queue (2Q), an algorithm using two queues to simulate LRU-2 algorithm with a constant time complexity. The algorithm places the pages with one reference in a special FIFO buffer. They will be move to the main LRU buffer only when the pages are accessed for the second time. They argued the cold pages in the FIFO buffer should be more easily evicted. Thus, the algorithm simulated the LRU-2 algorithm.

Park et al. [13] argued that traditionally replacement algorithms were customized for memory-based systems, but when it came to flash memory, the problem would be complicated by the difference in the read and write cost. As a result, they proposed Clean-First LRU (CFLRU), which considers both hit ratio and the cost to evict dirty pages to flash memory. It splits the cache into clean-first region and working region. The algorithm preferentially evicts pages in clean-first region to prevent writes. Their experiments showed reduced latency and lower energy consumption.

Blankstein et al. [6] proposed Hyperbolic, which has a sampling-based eviction policy with lazy evaluation. They statically model the requests based on a sequence of static distributions in which each distribution serves a period of time consecutively. Within a distribution, they evaluate each item by its frequency and time since entered the cache. This could avoid the over-punishment on new items in LFU.

Beckmann et al. [4] proposed LHD, a cache design aiming at maximizing hit density. Hit density is defined as hits per space consumed. An optimal algorithm should achieve occupying the whole Time-Space graph with items that are going to be hit again.

The eviction is based on object size, hit probability, and expected time in cache since we cannot know the access pattern in advance.

In addition to the researches purely on the caching algorithms, there are also many works focused on building other systems like storages or CDNs that also include improvements on cache designs.

Fan et al. [9] proposed MemC3, which are engineering improvements on Memcached. It leverages optimistic cuckoo hashing, a compact LRU-approximating eviction algorithm based on CLOCK, and optimistic locking. They argued that the optimistic cuckoo hashing algorithm could achieve space efficiency, cache-friendly, as well as reader concurrency. They also claimed that optimistic locking would increase concurrency via avoiding inter-thread synchronization.

Lim et al. [17] proposed MICA, an in-memory key-value storage. They would like MICA to achieve a great throughput and cache efficiency. Given that the LRU algorithms queue-based implementation would break memory cache locality, they used an approximation by only considering reads from corresponding cores. They argued the approximation is correct and effective.

Berger et al. [5] proposed AdaptSize, which is an adaptive, size-aware cache admission policy for Hot Object Cache (HOC) of a CDN network. They argued that the caching policy for HOC is extremely difficult due to the variation in request patterns and object sizes. Their method uses a Markov model for tuning to achieve improve the cache admission policy.

There are many works trying to leverage the correlation between different objects for caching. These algorithms mostly require additional knowledge such as the dependency DAG among the objects.

Yu et al. [27] proposed Least Reference Count (LRC), which considers the data dependency that occurs in systems like Spark, Tez, and Piccolo. The data dependencies in these systems can be represented by a directed acyclic graph (DAG). This algorithm evicts the objects whose reference counts are the smallest. They argued LRU wouldnt perform well in this scenario according to the data analysis of data generation and data transition to inactive, which shows newly generated data can be rapidly deprecated.

Yu et al. [26] argued in data-parallel environments like Spark, Tez, or Storm, optimizing cache hit ratio might not necessarily lead to faster completion time of tasks. They argued a task could be speed up only when all its input blocks were already cached. They proposed a metric called effective cache hit ratio, which counts the ratio of cache hits that can speed up a task. They also proposed Least Effective Reference Count (LERC), an algorithm that persists all the input blocks as a whole. They argued that analysis showed LERC could improve effective cache hit ratio comparing to LRC and LRU. They also claimed LERC could speed up tasks comparing to LRC and LRU.

Katta et el. [14] Proposed CacheFlow, a dependency-aware caching system for forwarding rules in SDNs. They argued the caching problem would be complicated when it comes to forwarding rules, in which the rules have overlapping patterns. They argued that algorithms used in CacheFlow should be able to avoid the problem caused by partial caching and optimize hit ratio.

Perez et al. [20] argued in a data-parallel scheme like Spark, LRC could improve the caching efficiency comparing to LRU because it leverages the dependency of data blocks. However, they also argued that the policy should also consider the space and time a data blocks could consume. They proposed Most Reference Distance (MRD), which is a cache management policy leveraging the DAG information among data blocks, from which the relative stage distance of each data block can be extracted. It evicts the furthest and the least likely data blocks. It also prefetches the nearest and most likely ones.

Yang et al. [25] adopted a mathematical way to devise an optimized algorithm for the caching policy of the [19]data-parallel scenarios like Spark. They reduced the NP-hard problem to a convex optimization problem which can be solve in linear time. They claimed the algorithm to have a hit ratio 12% higher than LRU.

3 Project Design

3.1 Analysis

When users access Xbox.com, many individual objects from multiples backends will be requested until the complete pages can be shown. A cache system is integrated with this system in order to leverage locality to reduce end-to-end latency.

Given that the system has a high fan-out architecture, the latency of a single request depends on the highest latency among the objects. Thus, achieving a higher page-level hit ratio is much more important than the object-level hit ratio. The page-level hit ratio is the portion of requests in which all the objects were hit in the cache, while the object-level ratio is the portion of single objects that were hit.

Our work differs from the existing work based on the assumption that the object accesses are independent with other objects and are aimed at the goal to improve the object-level hit ratio instead of the page-level hit ratio. Previous analysis shows the objects contribution to page-level hit ratio isnt proportional to its popularity. As a result, we would like to design the cache in a way that leverages both the popularity of the objects and the relationship between the objects and requests.

There are many existing works about the design of replacement algorithms. Many established systems use Least Recently Used (LRU), or Least Frequently Used (LFU). There are variants that leverage a longer history of accesses such as LRU-K. There are also algorithms considering object sizes as a factor of admission or eviction, such as LRU-MIN and LRU-THOLD. Some other algorithms consider the time since an object entered the cache, such as GD-S and Hyperbolic. There are also algorithms considering network bandwidth and retrieval costs such as LRV. Because our algorithm should still consider the popularity of the objects, we can still learn from the existing algorithms as the basis of our own. First of all, we are considering the frequency of the accesses to an object. We are also considering the size of objects to avoid giant objects occupying the cache spaces. We can use the methodology in LRU-THOLD for the admission policy of the cache. However, we can also integrate the size in the priority evaluation of the eviction process, if we choose not to implement an admission policy.

We also want the cache to consider the relationship between different objects in a request. Thus, we are keeping information about the corresponding requests with the objects. We memorize the sizes of the objects in requests, and the size of the missing objects in requests with each object. Considering this information may differ when an object belongs to multiple requests, we adopted a aging algorithm that update the information every time with a weight to combine with the previous values.

Recently, there are works emerged using sampling-based eviction policies that could approach Queue-based algorithms and significantly improve throughput. Hyperbolic and LHD are examples that uses sample-based algorithms that achieved better results than previous works. We found our cache also adaptable to this approach because the priorities can be evaluated just in time. This approach can also avoid a queue-based implementation, which could prevent us achieving a higher throughput due to the complex locking scheme.

We decided to implement our cache in Go. There are several benefits from using the language. First, Go is well known for the capability of developing large-scale systems with high throughput. Its goroutine mechanism enables developers to develop highly concurrent programs without pain. Go programs are also portable to multiple platforms while keeping the efficiency as they could run statically like C or C++. Go also provides garbage collection which helps developers to avoid memory-leakages. However, this could also bring difficulties when we implement our cache system. Since we cannot know the implications during memory allocation or deallocation process of Go, efficiency bottlenecks could be ignored and problems like memory fragmentation could happen. To avoid these problems, an object pool could be used to manage the occupied and free spaces by ourselves. The object pool should be layered by sizes of objects. It should also conduct lazy allocation and deallocation to keep most of the memory operations inside.

3.2 Architecture

3.2.1 Overview

Our system is implemented based on Ccache [1] an LRU cache in Go. A system provides a general architecture for object store. The system contains three major components, entry points, buckets, and LRU worker. The entry points are the APIs including Get, Set, Fetch, Replace, and Delete. These requests are redirected to buckets, which are the units that store the objects. A bucket is also a basic unit for locking. The APIs also generate a job each time for the LRU worker to properly manage the LRU queue. Once the worker detects an overflow of the cache space, it will conduct the eviction process to free up enough space for new objects to be inserted.

Before implementing our system, there is an outstanding problem that needs to be solved. The Ccache is essentially not scalable because of the implementation of LRU. Each API call will lead to a job to the LRU worker, which manages the LRU queue in a single thread to avoid race condition. The worker is also in charge of all the evictions. Only the objects could be concurrently accessed through different buckets. To parallelize the system more thoroughly, we ditched the worker that maintains the queue. We make the thread that conducts the API call to perform the eviction when necessary to enable fully concurrent execution. In our implementation, the data structures that need to be protected become really simple because the sampling-based eviction algorithm. Thus, we dont need to keep a centralized queue to store all the priorities of the objects to facilitate eviction process. We only need to sample several objects from the buckets each time which can be fully parallelized to each thread. The only variable that are shared between the threads are the total size of all the objects in the cache, which is updated during Set, Delete, and Replace. Because accessing the variable is simple and cheap, the contention should be very much reduced comparing the original worker implementation. We decided to use atomic operations to fetch and update the variable to even further reduce the contention.

To make the system compatible with page-level requests, we made several changes. First, two APIs, **SetPage** and **GetPage**, were added to the system with supporting functions. These two APIs works on all objects within one page-level requests at a time. These two APIs are necessary because the system needs to store information at page level in addition to information in object level, such as the summed size of all objects in a request. The page-level information can be collected and processed through these two new APIs and be stored to the buckets. Besides these two APIs, we also added an API called **SetPageWithMissingSize**. This interface is identical to **SetPage**, except it also takes the summed size of all missing objects from a **GetPage**. We want this information because its considered in one of our new eviction algorithms. Unlike the first two, this new API is not a must-have since the information could also be acquired within **SetPage**, but this implementation will introduce redundant computation given that the information is already provided by **GetPage**. Furthermore, we cannot make any guarantee about the calling sequence of the APIs since many threads are calling concurrently. Thus, we cannot store this information in any internal data structures. As a result, we decided that its worthy adding **SetPageWithMissingSize** to efficiently provide all the information we need to the caching system.

Each thread can call any APIs at any time with any order. When a cache overflow is detected by any thread, it is able to conduct a sampling-based eviction when other threads are still accessing the cache, or even performing an eviction simultaneously. In out implementation, the cache should be able to be shared between threads without a dedicated worker for eviction. In a single eviction process, several objects are sampled from all objects with equal probabilities. Their priorities are evaluated by the eviction algorithm. Then the object with the lowest priority value is evicted. This process should be repeated until enough space has been freed. To prevent eviction to happen too frequently, we are evicting at least 10 objects at a time as default. The minimum evicted object count is configurable.

3.2.2 Sampling

Sampling is an important feature in our system. It improves the scalability compared to the traditional queue-based algorithms since each thread is able to

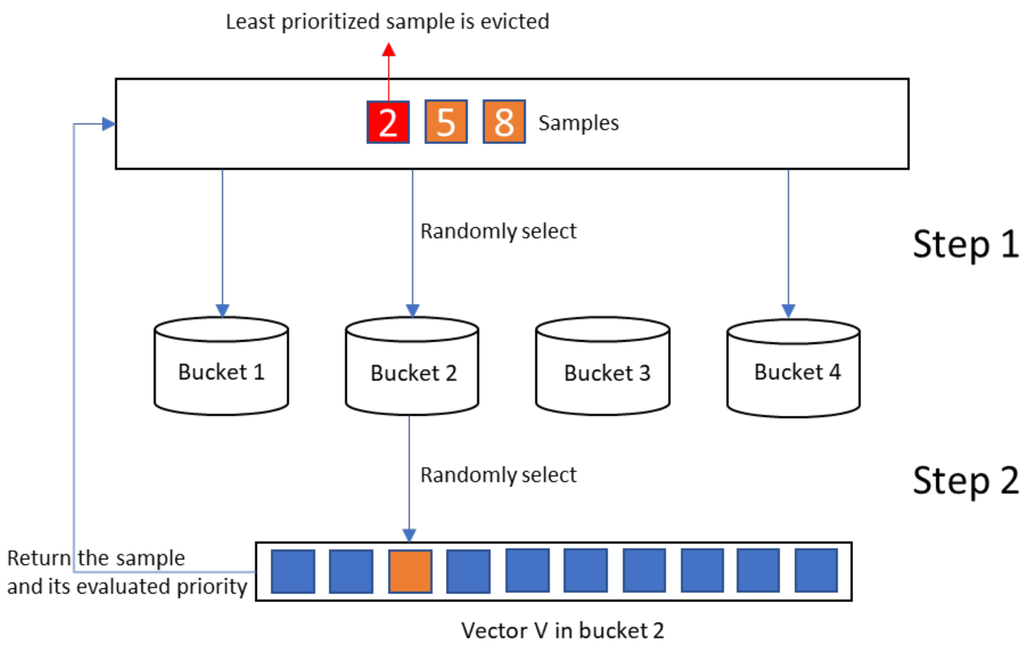

Figure 1: Sampling and Eviction

sample on its own. We designed a two-step sampling mechanism, to implement the sampling-based algorithms in an efficient and correct way. This process can sample one object each time, so it needs to be repeated until we have acquired enough samples. We conduct the sampling in two steps, which is shown in Figure 1.

The first step is implemented by the Alias Method [23]. This method has a pre-processing time of O(n) and a sampling time of O(1) for each sample. In the method, a probability table U and an alias table K are pre-processed before sampling. Each of the table is computed in O(n) time. To keep the size of both table reasonable and the content of the tables less dynamic, at this step we only sample buckets. Thus, the dimensions of both tables are only the number of buckets instead of objects, which reduces the pre-processing time. During experiments, we found that the objects are distributed very evenly to buckets, which means the probability for each bucket to get sampled should almost be equal and should not vary very rapidly. Considering this observation, we further reduced the pre-process computation by only updating the tables after 1,000 changes. This number is configurable to achieve better accuracy when needed. The U and K tables are shared between every thread. Thus, we protect both tables by only updating via compare-and-swap (CAS) to avoid race conditions and reduce contention. In this manner updates do not have to wait for readers to finish before updating the tables. After each update, new readers will have the newer version while old readers can still finish their sampling using the old readers, without interfering each other. To guarantee the correctness of the sampling, we sample one bucket from all buckets for each sample.

The second step is to sample one object from the selected bucket. As we have described above, a bucket is the basic unit for locking. All the operations in a bucket happen in single thread. We keep both an object vector **V** and a hashmap **H** from key to the object index in **V** in each bucket. During sampling, we randomly select one object from the vector **V**. For insertion, we append the object at the end of the vector **V** then update the corresponding entry in **H**. For replacement, we only need to get the index from **H**, then update the corresponding place in **V**. Under this implementation, our sampling, insertion, and replacement all have O(1) time complexity. To also achieve O(1) complexity in deletion, we need to first get the index from **H**, replace the place by the last object in **V**, then update **H** correspondingly and delete the last object in **V**.

The number of samples is configurable in our system. To get an estimate of the proper range, we consider existing research and production systems in the field. An interesting phenomenon was observed during the process. Previous research works that use sampling-based algorithms line LHD and Hyperbolic tend to use as many as 64 samples, while Redis, a production key-value store that also uses sampling-based eviction has a default configuration of 5 samples. We conducted several experiments to reveal the reasons behind this phenomenon, which will be further explained in Section 4.5 and Section 5.

After getting a sample from these two steps, it will be evaluated by an evaluation algorithm to decide if it should be evicted. However, evaluating the samples after they have been gathered require addition locking on the objects to avoid race conditions. This could complicate the system and create unnecessary latency. To prevent this unsatisfactory design, we implement the evaluation inside the second step in the buckets, to make the sampled objects be returned with its evaluation value. Then the comparison happens in each thread wont cause any race condition while doing eviction concurrently.

3.3 Evaluation Algorithms

The evaluation algorithms are important components this project focuses on. There are widely-used algorithms like LRU and LFU which are also compatible with sampling-based implementation. There are also most recent works that were directly developed with sampling, such as Hyperbolic and LHD. We expect our system to support LRU, LFU, and Hyperbolic for not only experiment purposes, but also for versatility in actual use. In addition, we also designed two new algorithms that specifically optimize PHR instead of OHR, we call them Simple Page-level Optimizer (SPO) and Sampling-Robust Page-level Optimizer (SRPO). Each algorithm is represented by an evaluation function that can be plugged into the sampling process to return a value of which the larger one represents higher priority.

To implement LRU and LFU in a sampling manner, we need to record additional information with each object in the buckets. For LRU, the information is the last access time, while for LFU, the information is the total access count. The evaluation function for LRU is the UNIX timestamp of the access time. The function for LFU is simply the access count. With these two functions, the sampling-based algorithms should approximate the queue-based LRU and LFU.

Hyperbolic [6] is a caching algorithm that was intrinsically built in a sampling manner. We implemented the cost-aware version of Hyperbolic, which requires the information of object size s, access count n, and the time since the object entered cache t, which can be calculated by the current time and the entered time that was stored with the object in cache. The priority function to evaluate object i is as Formula 1.

$$p_i = \frac{n_i}{s_i \cdot t_i} \tag{1}$$

Our new algorithm SPO was inspired by the idea of Hyperbolic to consider the frequency, object size, and time. We expect this algorithm to be tailored for the page-level requests instead of object-level requests that all traditional algorithms were focused on. Intuitively, we hope one request more likely to be fulfilled if there are fewer objects are missing, which means we can improve page-level hit ratio with lower cost. We realize the cost of the cache is actually the size, not the number, of objects to cache. Thus, we decided to use the total size of missing objects in one request instead of the object count to evaluate the priorities of the objects. The function to calculate the priority for object i from request j is as Formula 2.

$$p_i = \frac{n_i}{(\sum_{k \in req_j}^{k\ is\ missing} s_k) \cdot t_i} \tag{2}$$

There is a key implementation challenge in our system, which is persisting the request-related information with each object in the buckets. However, one object could exist in more than one different page-level requests. Thus, we need to update the information stored in a way that combines different sources, instead of simply resetting to the latest values each time. We decided to introduce a updating weight w between 0 to 1, which is used to update values as below. This weight is used to update the sum of missing size that is used in SPO. Formula 3 shows how we use this weight to update the information.

$$v = w \cdot v_{old} + (1 - w) \cdot v_{new} \quad (3)$$

In addition to SPO which leverage the properties of page-level requests at the evaluation of priority, we also want to examine if an admission policy that also related to page-level information can help the system improve the page-level hit ratio. Thats why we design SRPO on top of SPO. In this algorithm, the missing objects of a request wont be admitted if the cache is full and the sum of the missing object exceed a threshold. If the threshold is close to zero, the cache will barely take any new object if its already full, then the hit ratio should be very low. If the threshold is too large, it should approach the performance of SPO. We expect it could show better result than SPO when its been set properly. We conducted experiments to decide the best threshold for evaluation, which will be detailed in the Experiments section.

4 Experiments

4.1 Objectives

The goal of the experiments is to evaluate two main factors of our cache design, performance and efficiency. These two factors reflect the quality of our implementation and effectiveness of our algorithms.

To evaluate the performance, we conducted experiments measuring the page-level-hit-ratio (PHR) and the object-level-hit-ratio (OHR) given a specific workload. Considering the purpose of this project is to optimize PHR instead of OHR, our algorithms may not perform the best in terms of OHR, because the tradeoffs between these two metrics could differ, but an improved PHR is what we expect to see.

To evaluate the efficiency of our system, we will measure throughput. This metric could reveal different aspects of our implementation and algorithm design, such as the time complexity of a fetch or insert operation, the complexity of an eviction, etc. This metric should also show the scalability of the system on workloads and computing resources.

4.2 Environment and Settings

The platform for our experiments has Intel Xeon CPU E5-1620 v4 @ 3.50GHz with 4 cores and 8 hyper-threads. It comes with 32 GiB memory and SSD storage. It also has a swap space of 64 GiB. The platform runs Ubuntu 16.04.1, with a Golang runtime go1.10.4 linux/amd64.

The trace we are testing against for hit ratios was collected on 03/06/2018 with 30,000,000 page-requests, each contains one or more object-requests. The trace for throughput tests was also collected on 03/06/2018 with 1,000,000 page-requests. We select a larger trace for hit ratios because usually the metrics will not stabilize at the beginning of the trace. For throughput experiments this phenomenon should not be a problem.

In the later experiments, we define the cache sizes using portions of the working set of a trace. The definition of working set is the minimum size to store all the objects in a trace, in which the same object is only counted once. For the trace of 1,000,000 requests the working set is 36,078,002,798 bytes. The working set for the trace of 30,000,000 requests is 464,680,120,190 bytes. We consider 1% of the working set is a reasonable size of a caching system.

The system is configured using 128 buckets and the minimum number of evicted objects in each eviction is 10.

4.3 Preliminary Experiments

There are some preliminary results we need to have to decide several settings for later experiments. First of all, we need to decide the part of the trace we want to use to calculate the hit ratios. As we discussed above, the beginning of the trace should not be considered for calculation, even though the whole trace is replayed. We expect the metrics on the considered part of the trace to be stable. To get this preliminary result, we

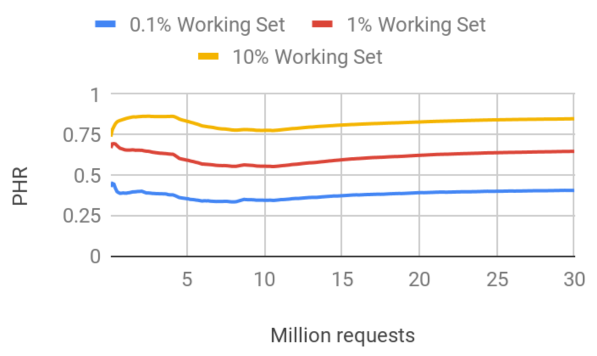

Figure 2: Trace Cutoff

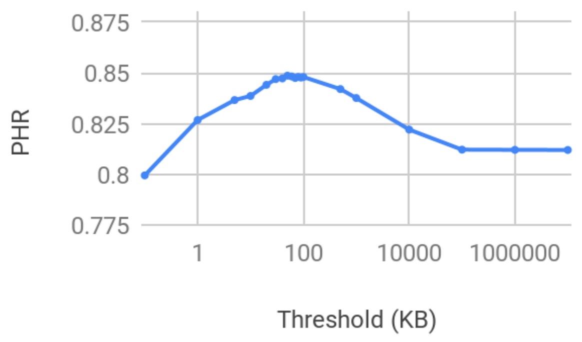

Figure 3: Admission Threshold

tested the cache using sampling-based LRU of 64 samples, of which the sizes are 0.1%, 1%, and 10% of the working set, which align with the settings we are using in the later experiments. We are using the trace of 30,000,000 requests, which is also the one we will use for hit ratio experiments. The metric we are measuring is PHR. This metric is not exactly the one that we want to measure in the later experiments because it is calculated from the beginning, but it is an effective indicator if the performance the system is already stable. We also expect OHR to have similar fluctuation on the same trace. Thus, we can just use the same cutoff that is decided based on this preliminary experiment.

The result is shown in Figure 2. As we can see, the metric PHR has significant fluctuation in the first 10 million requests. Then it gets stable without drastic change until the end. To keep enough data for measurements and eliminate all fluctuating part of the trace, we decide to set the cutoff to be 25 million, which means we perform the first 25 million requests without measuring, then count misses and hits in the last 5 million requests to calculate the hit ratios. This cutoff applies to all later experiments concerning OHR and PHR.

Another preliminary result we want to have is the admission threshold for algorithm SRPO. For this purpose, we conducted an experiment measuring PHR, with a cache of which the size is 1% working set, and the sample number is 32. The trace is the one with 30 million requests. The result is shown in Figure 3. At the beginning the PHR grows with threshold size, and then decreases and approaches a value around 0.81. This result matches our expectation. When the threshold is too small, few objects are allowed in when the cache is full. The cache should perform badly at the point. When the threshold is too large, it should approach the result of SPO. We found that the algorithm produced the best result when the threshold is around 100KB. We decide to use this value (100KB) as the threshold we are using for SRPO in the later experiments.

4.4 Design

4.4.1 Performance

As mentioned, the metrics we would like to measure to evaluate performance are OHR and PHR. The baselines of the experiments are LRU, LFU, and Hyperbolic. We choose LRU and LFU because they are widely used in similar systems and LRU is using in the existing applications servers of Xbox and also in the original Ccache. We also choose Hyperbolic because it represents state-of-the-art works in caching algorithms. In all the experiments, we use sampling-based versions or variants of the algorithms for consistency.

There are two variables that we are curious how they could affect the metrics. They are the cache size and the number of samples we are taking for each eviction. According to preliminary analysis, we choose to the algorithms on a cache of which the size is 0.1%, 1%, and 10% working set. As we mentioned, 1% working set is a decent size for a caching system, while 0.1% and 10% working set can simulate the situations where the cache is too small and too large. We consider these three values could provide enough reference for cache settings. We also want to test the performance of the algorithms with different sample numbers. The pattern that the metric shows on this variable could indicate the stability of the sampling-based algorithms when the sample number changes. This could also help users tradeoff between cache scalability and performance by choosing a specific number of samples. We choose to use 4, 8, 16, 32, 64 samples in the experiments. The reason we select this range is as what we mentioned above, we found there is a discrepancy among the numbers of samples that are chosen by research and open-source projects. In Redis, the default sample number is 5, while in LHD or Hyperbolic, the authors chose 64. We have preliminary results show sample number is a crucial factor that affects the eviction efficiency. We think this discrepancy represents different tradeoffs that the teams made. The research projects could focus more on hit ratio while the projects aiming at production should focus more on throughput. We hope our experiments can reveal the reason why these systems have such different setting of sample number.

Unlike the traditional algorithms without sampling, the results of this experiment from multiple runs of the same setting can be different from each other due to the randomness of the sampling process. To reduce the noise in the results, we repeat the same experiment for three times and average the hit ratios as the final results.

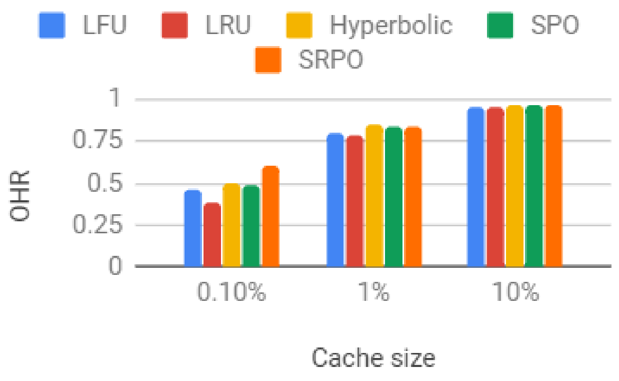

Figure 4: OHR with 4 Samples

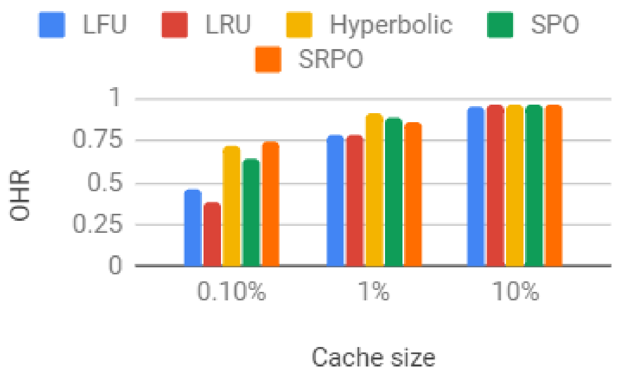

Figure 5: OHR with 64 Samples

4.4.2 Throughput

Throughput is an important metric for this project because this reflects if we are designing and implementing the system in a scalable manner. We would like our system to scale when we are providing more computing resources like CPU cores. This could prove our system is capable to serve the workload in production. To evaluate throughput, we would like to conduct two experiments. In this section, we only use SRPO in the experiments.

We are interested in how much three different variables can affect the throughput. They are the number of threads that are concurrently accessing the cache, the number of samples, and the cache size. To be consistent with previous experiments, we will use 4, 8, 16, 32, 64 samples, and cache size of 0.1%, 1%, 10% working set. We also choose to test with 1, 2, 4, 8 as the number of threads, which could potentially use all physical threads in the CPU. Ideally, the throughput should be proportional to the number of threads when its under the limit of physical threads. However, there could be contention between threads and when insertion and eviction happen, which undermines the scalability of the system. We are wondering if the stability of the system could also be affected by the number of samples that we are using in SRPO.

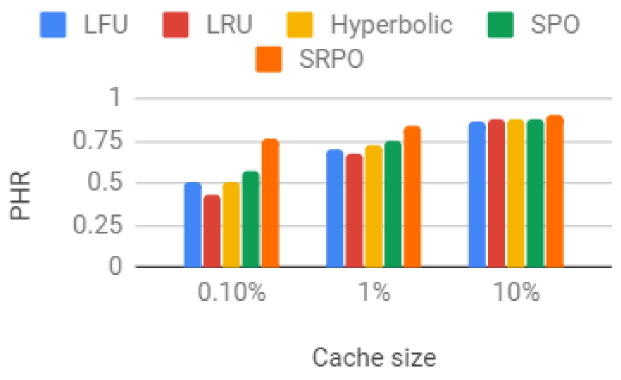

Figure 6: PHR with 4 Samples

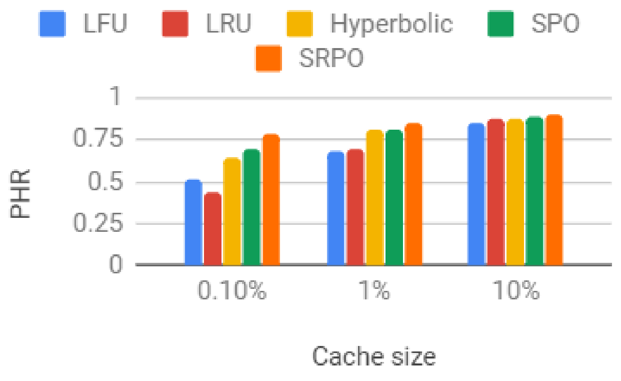

Figure 7: PHR with 64 Samples

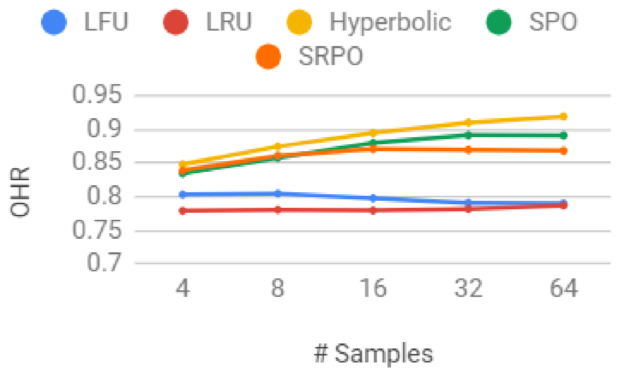

Figure 8: OHR and Sample Number

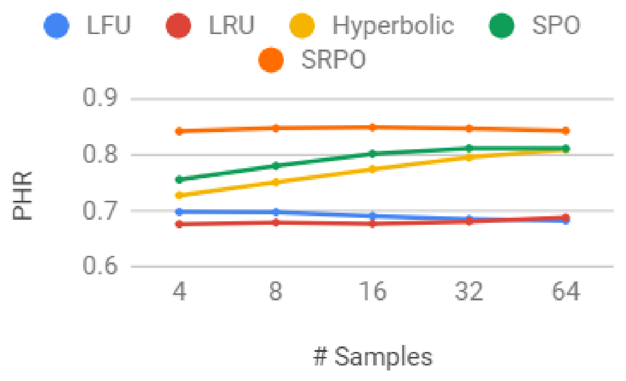

Figure 9: PHR and Sample Number

4.5 Results

4.5.1 Performance

Figure 4 and Figure 5 are the results showing how OHR changes with the cache size. Figure 6 and Figure 7 are the results showing how PHR change with the cache size. Figure 4 and Figure C are the results with 4 samples while Figure 5 and Figure 7 are the results with 64 samples. As we can see, OHR and PHR increase in all cases when cache size grows. We can observe among different algorithms, the performance differs when the cache size is 0.1% or 1% working set, while when the size is 10% the differences are not significant. This is probably because the cache is relatively over-provisioned that most of the misses are compulsory. Then all algorithms perform roughly the same. When we are looking at OHR, we can see that Hyperbolic, SPO, and SRPO have better results than LRU and LFU. We can also see that the performance of Hyperbolic and SPO changes significantly when the sample number varies. Hyperbolic performs better than our algorithms, SPO and SRPO, when the sample number is 64. This is a reasonable result because Hyperbolic is specifically designed for object-level requests while our algorithms are designed for page-level requests instead.

When it comes to PHR, similar patterns can also be observed. Hyperbolic and SPO also perform very differently when the sample number changes. But for this metric, SPO and SRPO out-perform Hyperbolic and other algorithms consistently when cache size is 0.1% and 1% working set. This should our algorithms can improve page-level hit ratio compared to traditional

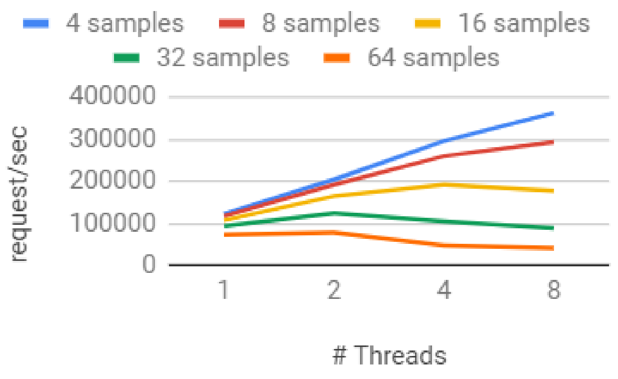

Figure 10: Throughput with 0.1% Working Set

algorithms like Hyperbolic, LRU or LFU.

To better reveal how stable the algorithms could perform the when the sample number changes, we have Figure 8 and Figure 9. These two figures demonstrate how OHR and PHR changes with sample number accordingly. All the experiments in these two figures were done with cache size of 1% working set. From the figures, we can know that for OHR, the performance of Hyperbolic, SPO, and SRPO is not very improves relatively significant compared to LRU and LFU. This indicates these algorithms require a large sample number to have a better OHR. For PHR, the performance of Hyperbolic and SRPO still changes very drastically when the sample number changes. But for SRPO the situation is quite different. It performs consistently well despite the sample number is as small as 4. We call this property sampling-robust. We can also call LRU and LFU sampling-robust for both OHR and PHR, even though their performance is not comparable to Hyperbolic, SPO or SRPO. This is an ideal property for a sampling-based algorithm, because an algorithm with this property can still well function when the sample number is small. The sampling-robust property of LRU can also explain why Redis can use only 5 samples as default, because it should be able to produce satisfactory result.

4.5.2 Throughput

Figure 10, Figure 11 and Figure 12 show how throughput changes with number of threads and number of samples with cache size of 0.1%, 1%, and 10% working set accordingly. All three figures have similar pat-

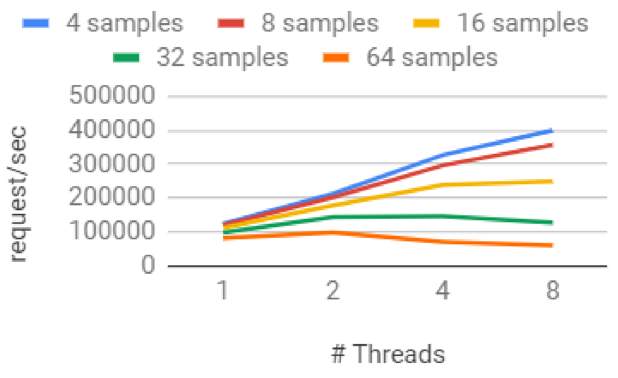

Figure 11: Throughput with 1% Working Set

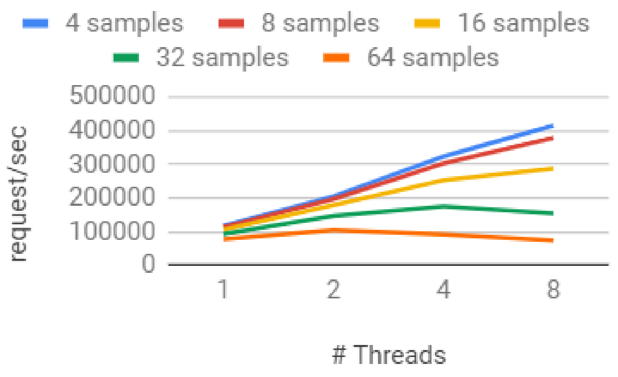

Figure 12: Throughput with 10% Working Set

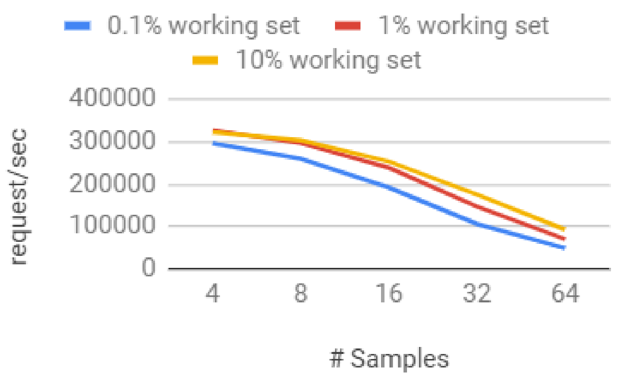

Figure 13: Throughput and sample number

tern, which is when the sample number is smaller, the throughput increases closer to linearly with number of threads. When the sample number is larger, the throughput increases more slowly, then decreases. Figure 13 shows the result of throughput of 4 threads. It shows the general trend how throughput will change when the sample number increases. As we can see, the throughput decreases when the sample number increases. These figures show that the scalability of the system could be hurt by having too many samples for one eviction process. Our hypothesis on this phenomenon is that the eviction cost and synchronization overhead grow as the sample number increase.

5 Conclusions

We identified serving page-level requests is a new problem for caching policies different from object-level requests. Solving this new problem requires both changes in cache API and architecture, as well as the algorithms. The cache system should support operations including multiple objects in a single request. It also needs to store page-level information in addition to object-level information that traditional caching algorithms use.

We designed two algorithms, SPO and SRPO, to solve this new problem. Both of them consider object-level information such as frequency and time, with page-level information such as the size of missing objects in a request. Furthermore, SRPO includes an admission policy that exclude requests with too large missing objects when the cache is full.

We conducted experiments showing the relationship between performance of SRPO and its admission threshold to select the optimal one for later experiments. Our experiment results also show SPO and SRPO could outperform all other algorithms, including LRU, LFU and Hyperbolic in PHR. They also perform decently in OHR, even though Hyperbolic sometimes has better result. Our experiment results also indicate SRPOs performance in PHR rarely fluctuate when the sample number changes. We call this property sampling-robustness. We consider this a ideal property for a sampling-based algorithms because an algorithm with this property should be able to produce similar results under different sampling settings.

Our throughput experiment results demonstrate that the system scales near linearly when the sample number is small, but it could barely scale when the sample number is too large. They also show that the system consistently performs better when the sample number is small under the same settings. This phenomenon could explain our observation of the discrepancy of sample number choices between Redis and research systems line LHD and Hyperbolic, because for non-sampling-robust algorithms like LHD and Hyperbolic, there is a tradeoff between the performance and scalability, but for sampling-robust algorithms like LRU that Redis uses, it is able to only consider the throughput.

6 Future Work

There are two types of work that we consider promising in the future. First, new algorithms could be designed for the page-level requests in the future. It is reasonable for them to consider more page-level information as well as object-level information. They can also innovate on admission and garbage collection policies. New algorithms can also focus on improving scalability of the system. According to our throughput results, it is possible that changes on the sampling mechanism could reduce its effects that influence the scalability.

Second, sampling-robustness is a very interesting topic because it doesnt show any pattern so far.

We can tell that simple algorithms like LRU and LFU are sampling-robust, while more sophisticated algorithms like Hyperbolic and SPO are not. But SRPO is apparently not a simple algorithm but it is sampling-robust. We definitely hope to have more insights about which parts of the algorithm contribute to its sampling-robustness, and which parts are not.

References

[1] godoc package ccache. https://godoc.org/github.com/karlseguin/ccache. Accessed: 2018-12-21.

[2] ABRAMS, M., STANDRIDGE, C. R., ABDULLA, G., WILLIAMS, S., AND FOX, E. A. Caching proxies: Limitations and potentials.

[3] ARLITT, M., CHERKASOVA, L., DILLEY, J., FRIEDRICH, R., AND JIN, T. Evaluating content management techniques for web proxy caches. *ACM SIGMETRICS Performance Evaluation Review 27*, 4 (2000), 3–11.

[4] BECKMANN, N., CHEN, H., AND CIDON, A. Lhd: Improving cache hit rate by maximizing hit density. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)* (2018), USENIX

[5] BERGER, D. S., SITARAMAN, R. K., AND HARCHOL-BALTER, M. Adaptsize: Orchestrating the hot object memory cache in a content delivery network. In *NSDI* (2017), pp. 483–498.

[6] BLANKSTEIN, A., SEN, S., AND FREEDMAN, M. J. Hyperbolic caching: Flexible caching for web applications. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)* (2017), pp. 499–511.

[7] CAO, P., AND IRANI, S. Cost-aware www proxy caching algorithms. In *Usenix symposium on internet technologies and systems* (1997), vol. 12, pp. 193–206.

[8] EINZIGER, G., FRIEDMAN, R., AND MANES, B. Tinylfu: A highly efficient cache admission policy. *ACM Transactions on Storage (TOS) 13*, 4 (2017), 35.

[9] FAN, B., ANDERSEN, D. G., AND KAMINSKY, M. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *NSDI* (2013), vol. 13, pp. 371–384.

[10] HUANG, Q., BIRMAN, K., VAN RENESSE, R., LLOYD, W., KUMAR, S., AND LI, H. C. An analysis of facebook photo caching. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), ACM, pp. 167–181.

[11] JIANG, S., AND ZHANG, X. Lirs: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. *ACM SIGMETRICS Performance Evaluation Review 30*, 1 (2002), 31–42.

[12] JUSTINWANG, B., BERGER, D., AND SEN, S. Maximizing page-level cache hit ratios in large web services.

[13] KANG, J., KIM, J., PARK, S., JUNG, D., AND LEE, J. Cflru: a replacement algorithm for flash memory.

[14] KATTA, N., ALIPOURFARD, O., REXFORD, J., AND WALKER, D. Cacheflow: Dependency-aware rule-caching for software-defined networks. In *Proceedings of the Symposium on SDN Research* (2016), ACM, p. 6.

[15] KIM, J. M., CHOI, J., KIM, J., NOH, S. H., MIN, S. L., CHO, Y., AND KIM, C. S. A low-overhead high-performance unified buffer management scheme that exploits sequential and looping references. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4* (2000), USENIX Association, p. 9.

[16] LEE, D., CHOI, J., KIM, J.-H., NOH, S. H., MIN, S. L., CHO, Y., AND KIM, C. S. On the existence of a spectrum of policies that subsumes the least recently used (lru) and least frequently used (lfu) policies. In *ACM SIGMETRICS Performance Evaluation Review* (1999), vol. 27, ACM, pp. 134–143.

[17] LIM, H., HAN, D., ANDERSEN, D. G., AND KAMINSKY, M. Mica: A holistic approach to fast in-memory key-value storage. USENIX.

[18] MEGIDDO, N., AND MODHA, D. S. Arc: A self-tuning, low overhead replacement cache. In *FAST* (2003), vol. 3, pp. 115–130.

[19] O'NEIL, E. J., O'NEIL, P. E., AND WEIKUM, G. The lru-k page replacement algorithm for database disk buffering. *Acm Sigmod Record 22*, 2 (1993), 297–306.

[20] PEREZ, T. B., ZHOU, X., AND CHENG, D. Reference-distance eviction and prefetching for cache management in spark. In *Proceedings of the 47th International Conference on Parallel Processing* (2018), ACM, p. 88.

[21] RIZZO, L., AND VICISANO, L. Replacement policies for a proxy cache. *IEEE/ACM Transactions on Networking (ToN) 8*, 2 (2000), 158–170.

[22] STAROBINSKI, D., AND TSE, D. Probabilistic methods for web caching. *Performance evaluation 46*, 2-3 (2001), 125–137.

[23] WALKER, A. J. New fast method for generating discrete random numbers with arbitrary frequency distributions. *Electronics Letters 10*, 8 (1974), 127–128.

[24] WOOSTER, R. P., AND ABRAMS, M. Proxy caching that estimates page load delays. *Computer Networks and ISDN Systems 29*, 8 (1997), 977–986.

[25] YANG, Z., JIA, D., IOANNIDIS, S., MI, N., AND SHENG, B. Intermediate data caching optimization for multi-stage and parallel big data frameworks. *arXiv preprint arXiv:1804.10563* (2018).

[26] YU, Y., WANG, W., ZHANG, J., AND LETAIEF, K. B. Lerc: Coordinated cache management for data-parallel systems. In *GLOBECOM 2017-2017 IEEE Global Communications Conference* (2017), IEEE, pp. 1–6.

[27] YU, Y., WANG, W., ZHANG, J., AND LETAIEF, K. B. Lrc: Dependency-aware cache management for data analytics clusters. In *INFOCOM 2017-IEEE Conference on Computer Communications, IEEE* (2017), IEEE, pp. 1–9.

[28] ZHOU, Y., PHILBIN, J., AND LI, K. The multi-queue replacement algorithm for second level buffer caches. In *USENIX Annual Technical Conference, General Track* (2001), pp. 91–104.