CacheSack: Admission Optimization for Google Datacenter Flash Caches

Tzu-Wei Yang, Seth Pollen, Mustafa Uysal, Arif Merchant, and Homer Wolfmeister

Google Inc.

{twyang, pollen, uysal, aamerchant, wolfmeister}@google.com

Abstract

This paper describes the algorithm, implementation, and deployment experience of CacheSack, the admission algorithm for Google datacenter flash caches. CacheSack minimizes the dominant costs of Google's datacenter flash caches: disk IO and flash footprint. CacheSack partitions cache traffic into disjoint categories, analyzes the observed cache benefit of each subset, and formulates a knapsack problem to assign the optimal admission policy to each subset. Prior to this work, Google datacenter flash cache admission policies were optimized manually, with most caches using the Lazy Adaptive Replacement Cache (LARC) algorithm. Production experiments showed that CacheSack significantly outperforms the prior static admission policies for a 6.5% improvement of the total operational cost, as well as significant improvements in disk reads and flash wearout.

1 Introduction

Colossus Flash Cache (Figure 1) is the general-purpose flash cache service for Colossus [20], the successor to the Google File System [19]. Disk reads are expensive and are a major cost in datacenters: while disks are growing in storage capacity, the IO capacity (the ability to offer disk accesses per second, mainly disk reads) is not growing proportionally. As a result, to provision the IO requirements, we need to deploy a lot of hard disks that are not for storage but for the target IO capacity, which is costly.

Colossus Flash Cache provides a cost-effective way to improve IO capacity while costing a fraction of an equivalent RAM cache or deploying more hard disks. The primary design goal of Colossus Flash Cache is to reduce the amount of hard disk reads, in order to reduce disk IO requirements and costs. [1] Colossus Flash Cache serves the read traffic of many widely-used Google services including Colossus and database systems such as BigQuery [37], BigTable [11], F1 [34], and Spanner [13].

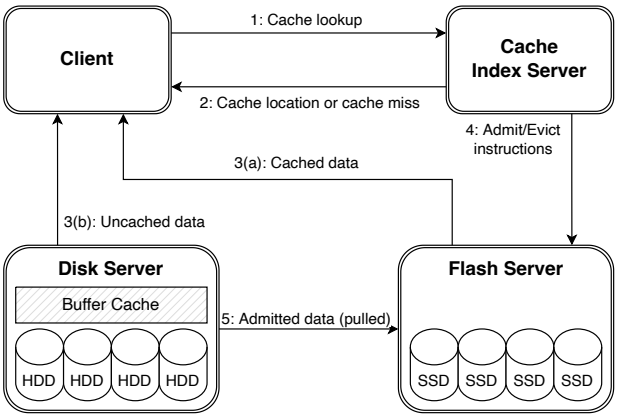

Figure 1: Colossus Flash Cache system.

CacheSack is the cache admission algorithm used by Colossus Flash Cache, intended to minimize the total cost of ownership (TCO). Compared to a RAM cache, a flash cache usually provides a much larger cache-to-storage capacity, and so a simple algorithm such as LRU may achieve a good cache hit-ratio. An idealized LRU is difficult to implement in a flash cache; we address this issue in Section 4. However, flash memory has limited write endurance, so may cause premature flash wearout and increase TCO. Write amplification and flash wearout, along with caching in Colossus disk servers, form a special challenge for designing a cache algorithm for Colossus Flash Cache.

2 Our contributions

CacheSack is the cache admission algorithm for Colossus Flash Cache, the successor to LARC (Lazy Adaptive Replacement Cache). CacheSack dynamically analyzes the cacheability of a workload and the given cache size, making the admis-

[1] While reducing read latency is also a desirable goal, it is not a design goal for Colossus Flash Cache, and beyond the scope of this paper.

sion decision for the workload. CacheSack was deployed in Colossus Flash Cache in May 2021 and is now Colossus Flash Cache's default cache admission algorithm. Our contributions are summarized as follows:

- CacheSack partitions traffic into multiple categories, estimates the disk reads and cost of write of each category, and formulates a knapsack problem that finds the optimal admission policy per category to minimize the overall cost, including disk reads and bytes written to flash.
- CacheSack effectively reduces the total cost of ownership (TCO) of Colossus Flash Cache. Compared to LARC, it results in 6% lower disk reads, reduces bytes written to flash by 26%, and improves TCO by 6.5% (one week average).
- CacheSack runs in real time, using a fraction of the resources of a cache index server.
- CacheSack is fully decentralized (as is Colossus Flash Cache). It requires only the information received by a single cache index server, and the failure of a single cache index server does not impact others.
- CacheSack supports major Google database systems and requires zero configuration if using these systems. For other applications, users only need to provide category annotations (Section 5.1).

3 Background

3.1 Write amplification

Non-sequential writes to a flash drive can cause serious write amplification [29, 42], a phenomenon where one logical write causes multiple physical writes. A flash byte has to be erased before it can be rewritten. A flash block is a continuous region of bytes in a flash drive and is the smallest unit that can be erased. As a result, a flash drive needs to move the live bytes in a flash block somewhere else before this flash block can be erased, which is called write amplification. Extra writes caused by write amplification reduce the IO performance and the lifetime of a flash drive; both greatly increase the cost of operating a flash cache.

Sequential cache evictions (like FIFO) result in large sequential areas that can be easily erased and reused later when admitting new data. By contrast, non-sequential evictions (such as those caused by LRU) result in a fragmented cache space and the flash drive has to move the interspersed live bytes somewhere else before erasing a block.

As a result, most existing eviction algorithms for RAM caches cannot be directly applied to flash caches, and write amplification is one of the most important factors to consider when designing a flash cache algorithm. Both Google [1] and Facebook [18] use FIFO-based evictions or other special purpose algorithms [36, 44] for production flash caches because of write amplification. Colossus Flash Cache reduces write amplification brought by non-sequential evictions by using *approximate* LRU (Section 4).

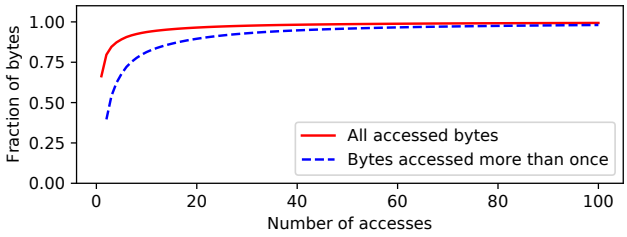

Figure 2: Fraction of bytes accessed a given number of times from disk plus flash over a week (right truncated at 100 accesses).

3.2 Write endurance

Flash has limited write endurance, and thus admitting all data into Colossus Flash Cache upon write or even upon the first read would wear out the flash too soon, significantly increasing TCO. To mitigate this issue, Colossus Flash Cache previously used Lazy Adaptive Replacement Cache (LARC) [21] to exclude data that are accessed only once by inserting data at the second access. Figure 2 shows that more than 60% of the traffic of Colossus Flash Cache is accessed only once, and so LARC can greatly reduce bytes written to flash and avoid cache pollution.

However, excessive flash writes are still possible with LARC, and as a workaround, Colossus Flash Cache used a write rate limiter to avoid an excessively high write rate. This is, however, a blunt approach, since it does not accurately factor in the impact on overall cost, and treats all workloads similarly. It may be preferable to allow some highly cacheable workloads to burst writes at the expense of other less cacheable workloads rather than throttling all write rates. CacheSack uses a more flexible and accurate approach by optimizing the total costs, including the write costs and the cost of disk reads.

3.3 Capturing second-access hits

LARC leverages the fact that a large fraction of data is accessed only once. Inserting data into cache only upon the second access avoids flash writes for data accessed once, reducing flash wear. However, the cost is that all second accesses are cache misses. Figure 2 shows that of the data accessed more than once in our workloads, 39% is accessed exactly twice, and these second accesses are cache misses under LARC. This has a significant performance impact, as was also observed in Facebook's cache for social network photos [36]. Our workaround for this when using LARC was to monitor the performance loss, and to manually turn off LARC (i.e., admit all data on the first miss) for workloads that suffered a significant performance penalty. However, the

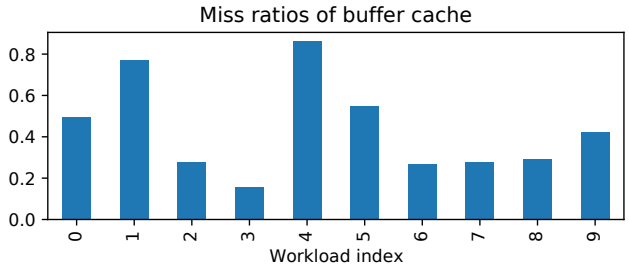

Figure 3: Miss ratios for 10 workloads at the disk server buffer cache if there is no Colossus Flash Cache (simulated).

manual maintenance to identify and set up special cases became more and more labor-intensive with the rapid adoption of Colossus Flash Cache in production. In our redesign, it was a requirement that the cache admission algorithm should be automatic and not require manual adjustments.

3.4 Colossus buffer cache

In addition to Colossus Flash Cache, Colossus maintains a RAM *buffer cache* in the lower level disk servers that buffers recent reads and writes as well as data prefetched from the disk. A cache miss in Colossus Flash Cache does not cause a disk read if the access hits in the buffer cache. Many Colossus workloads use the buffer cache extensively to improve IO performance.

In many cases, the cache hit ratio of Colossus Flash Cache is only weakly correlated with the actual disk read reduction, especially for workloads that are highly optimized for the buffer cache. Figure 3 shows simulated miss ratios of the disk server buffer caches with no Colossus Flash Cache for ten selected workloads, and they range from below 20% to over 80%. These miss ratios represent the upper bound on how far Colossus Flash Cache can improve the disk read rates. For workloads with low buffer cache miss ratios, hits in Colossus Flash Cache may simply replace buffer cache hits without improving the disk read rates. As a result, flash cache hit ratios are not a good metric to measure the efficacy of Colossus Flash Cache. In fact, our production results (Section 7.1) show that an admission policy can sometimes provide a higher hit ratio in Colossus Flash Cache but cause worse disk read rates.

3.5 Online and realtime requirements

Colossus Flash Cache is a fully decentralized system, so its cache algorithm can only use the resources of individual cache index servers, and heavyweight algorithms, such as machine learning (ML) models, may not be feasible. The binary of Colossus Flash Cache is updated on a weekly basis, while workloads change much more rapidly, so it is difficult for an offline-trained static model updated with the binary to adapt to workload changes. Therefore, we decided to use an online-trained model.

4 Overview of Colossus Flash Cache

Colossus Flash Cache consists of independent cache index servers. A cache index server does not directly hold cached data, but keeps an in-memory lookup table, called the index, that tracks the locations of cached data stored in the flash drives that reside on independent storage servers.

When a Colossus Flash Cache client requests to access data stored in Colossus, the client first sends an RPC to a Colossus Flash Cache cache index server (see Figure 1) to determine if the requested data are already cached on a flash server (a flash hit). If so, the cache index server sends back sufficient information for the client to access the flash copy of the data directly from the flash server. For a flash cache miss, the client contacts the disk server to read the data, while the cache index server independently decides whether to admit the data into the flash cache. If the cache index server decides to admit the data into flash, it instructs the flash server to pull the data from the disk server directly. The extra latency of communicating with the cache index servers is negligible compared to typical remote disk read latencies, and the latencies between remote flash reads and remote disk reads are in different orders of magnitude, so Colossus Flash Cache typically reduces overall latency, although this is not an explicit service goal. The goal is reducing TCO by avoiding expensive disk reads.

The *buffer cache* of a disk server also caches recently accessed data and prefetches a small amount of data into memory for a few seconds, so that reading recently-accessed data from an disk server does not necessarily cost extra disk reads. Colossus users are encouraged to design their workloads to improve IO performance by utilizing this buffer cache.

Colossus Flash Cache uses an *approximate* LRU eviction strategy to manage evictions. An idealized LRU cache would always evict the least recently used block from the cache when the cache is full. However, idealized LRU evictions cause non-sequential writes to flash, resulting in write amplification [29, 42]. To mitigate the issue of write amplification, Colossus Flash Cache uses evictions similar to Second Chance [30] to approximate LRU evictions: each Colossus Flash Cache cache index server manages a FIFO queue of many fixed-sized Colossus files (typically 1 GiB), each of which contains cache blocks. When evicting the file from the tail of the queue, we reinsert 28% of the most-recently-used blocks into the file at the head of queue. The percentage of the reinserted blocks is a tradeoff between the amount of hot blocks recycled, which improves the cache hit ratio, and the rate of reinsertion into flash, which increases write amplification. The current value (28%) is selected experimentally to strike a good balance between cache performance and write amplification. This way, the write amplification factor is effectively 1.28. A comparison of the performance of Second

Chance [30] indicates that the performance is quite close to that of LRU. Therefore, for ease of modeling, we approximate Colossus Flash Cache as an LRU cache.

Each Colossus Flash Cache server maintains a *ghost cache* [21], an in-memory lookup table that maps the key of data to the data's last access time, regardless of whether they are actually cached on flash. This is a key component of CacheSack, which relies on inter-arrival times to quickly build all the estimates described in Section 5.

Each cache index server represents a fraction of the key space, and one server's failure does not impact other cache index servers. To maintain the same reliability, CacheSack is also designed in the same decentralized manner: each cache index server runs its own CacheSack model, using only the information received by the cache index server, and its admission decisions do not affect other cache index servers.

5 CacheSack

5.1 Traffic partitioning

CacheSack partitions potential cache blocks into many *categories*, and assigns an admission policy to each category.

The majority of Colossus Flash Cache traffic comes from Google's database systems like BigTable and Spanner where categories can be well-defined. For database traffic, CacheSack defines a category as the combination of the table name, locality group [11, 13], and type for BigTable and Spanner, and a similar combination for other databases. Since Colossus Flash Cache is also available for other Colossus users, those users can define their own categories by annotating their data. If a user does not provide a category annotation, CacheSack will use the user name contained in the Colossus file path.

CacheSack then selects the right policy based on the pattern that category exhibits. Later, we will explain how we formulate CacheSack as a knapsack-like problem: given the cache capacity, how CacheSack chooses the items (categories) to minimize the overall cost.

5.2 Admission policies

We consider four admission policies that can be assigned to each category:

- `AdmitOnWrite`: Inserts a cache block at a write access or on any read cache miss.
- `AdmitOnMiss`: Inserts a cache block on any read cache miss, but does not insert a block at a write access. This is the conventional admission policy used in most of the cache literature.
- `AdmitOnSecondMiss` (LARC): Equivalent to Lazy Adaptive Replacement Cache (LARC); Inserts a block only after the second read access (miss), and only if the last access time is not older than the oldest last access time of the blocks in the cache, to reduce the insertion rate of cold blocks. LARC is scan resistant: any scanned data (accessed exactly once) will not be admitted.
- `NeverAdmit`: Never inserts blocks.

We can sort these policies by aggressiveness: `NeverAdmit` < `AdmitOnSecondMiss` < `AdmitOnMiss` < `AdmitOnWrite`.

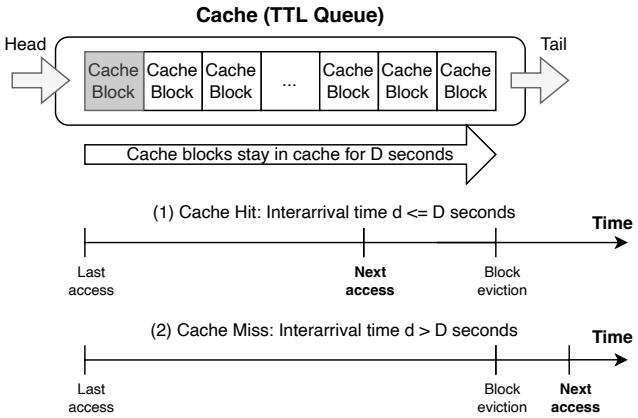

Figure 4: LRU evictions are approximated by TTL evictions with the modeled retention time D while the TTL counter of a cache block is reset whenever the cache block is accessed. If the access interarrival time d is less than or equal to D, this access is a cache hit and we move the cache block to the head of the queue (the TTL counter is reset). If the access interarrival time d is greater than D, this access is a cache miss and we insert the cache block to the head of the queue (the TTL counter is also reset).

5.3 Fast approximation to an LRU model

To determine the best policy for the cache, the most intuitive way is to simulate all possible policy-category combinations, which is a combinatorial knapsack problem (NP-Hard). Because CacheSack currently allows up to 5000 categories (Section 6.1) and uses 4 policies, there are up to 4^{5000} combinations and the knapsack problem can not be done even with downsampled traces. Instead, we use a fast approximation for modeling an LRU cache, by introducing the *modeled cache retention time*. The cache retention time is the maximum duration that a block stays in the LRU cache without any intervening accesses to it. In practice, the cache retention time varies slowly over time. Here we assume the *modeled* cache retention time is a constant D and this assumption will make all our estimates just approximations.

We use `AdmitOnMiss` as an example. For a given block, when a read access arrives, we can compute d, the time since last access (which is ∞ if the current access is the first read). We can classify the inter-arrival times by using D (Figure 4):

- $d \leq D$: An access arrives before the block leaves the cache, and therefore the access generates a cache hit and moves the block to the head of the queue.

- $d > D$: An access arrives after the block leaves the cache, and therefore it is a cache miss, which causes a write to the cache.

In other words, we approximate the LRU cache by a cache that has the TTL value D and resets the TTL counter of a block when receiving an access to the block. The theoretical aspect of the TTL approximation was also studied in literature. Fagin [17] showed the TTL approximation is asymptotically exact for independent and identically distributed requests, and [23] proved that given the assumption that data accesses are stationary and ergodic, the TTL approximation will converge to an LRU cache as the cache size goes to infinity. The accuracy of the TTL approximation in production is analyzed in Section 7.1.

A cache miss in Colossus Flash Cache will cause a disk read if it is also a miss in the Colossus buffer cache. Each cache index server maintains a *buffer cache simulator*, and when $d > D$, we run the simulator and see whether it is a miss.

This way, when a new access arrives, we are able to update the disk reads, cache usage, and bytes written to flash cache caused by admitting the block using `AdmitOnMiss`. We can also compute the same quantities for other policies: `AdmitOnSecondMiss`, `AdmitOnWrite`, and `NeverAdmit`. The detailed estimation is described in Appendix A.2.

A nice property of this approximation is that the estimates for a block are not affected by other blocks or policies, as long as the modeled cache retention time is given. Therefore, the disk reads, cache usage, and written bytes caused by admitting a category are just the sums of the corresponding block-level quantities.

5.4 Knapsack problem

Once we have the estimates for disk reads, cache usage, and bytes written to flash cache for each policy-category pair, we have a knapsack problem: find the optimal policy per category to minimize the overall cost (disk reads and written bytes) while fitting within the cache. We omit the definitions of the relative cost of disk reads, bytes written to flash, and flash storage, because they are confidential.

We further allow *fractional* policies: CacheSack can apply a policy to a fraction of a category. For example, CacheSack may decide it is optimal to apply `AdmitOnMiss`, `AdmitOnSecondMiss`, `AdmitOnWrite`, and `NeverAdmit` to 30%, 20%, 10%, and 40% of blocks in a category, respectively. Then the problem becomes a *fractional* knapsack problem [14] that finds the optimal policy fractions per category to minimize the overall cost. The advantage of considering a fractional knapsack is that it can be solved efficiently by a greedy algorithm, as opposed to a combinatorial knapsack that is NP-Hard. Our problem is slightly different from the original fractional knapsack in [14] because we need to decide four fractions per category instead of two. Appendix A.4 explains the details of how we solve our problem by a greedy algorithm after applying Andrew's monotone chain convex hull algorithm [2]. We note that if an LRU cache is perfectly modeled by the TTL approximation, the resulting cache retention time of the LRU cache is exactly D after applying the optimal policy fractions per category.

5.5 Optimization over modeled cache retention times

The knapsack problem in Section 5.4 is to find the optimal policy fractions for a *given* modeled cache retention time D, which can not be known in advance. Thus, we need to solve the same knapsack problem for *all* possible D. To do this in production, we can have a set of predefined modeled cache retention times: $0 < D_1 < D_2 < \cdots < D_m = \mathbf{D}$ where $\mathbf{D}$ is a suitable upper bound, and solve m different knapsack problems. Thanks to the greedy algorithm, we can still solve many knapsack problems (currently 127, Section 6.2) quickly.

6 CacheSack in production

CacheSack is now deployed in production as the default cache admission algorithm for Colossus Flash Cache. This section explains the engineering efforts needed to do so.

6.1 Category assignment

The number of categories encountered in production can not be known in advance, so we balance the need for accuracy and space by hashing a category to one of 5000 buckets. Categories assigned to the same bucket are treated as combined in the optimization. The number of hash buckets is a trade-off between memory usage and hash collisions. The typical number of categories per server is less than 100 and our experiments showed that with 5000 buckets, 95% of the clients see a hash collision rate lower than 1% and the worst collision rate is less than 5%. Further, cache collisions are not persistent, since each cache index server uses a different hash key and changes it periodically to break possible spatial and temporal correlations.

A bucket without sufficient training data might not provide meaningful metrics. If a bucket contributes to less than 0.1% of total lookups, it will be aggregated to a single *catch-all* bucket before solving the knapsack problem.

6.2 Modeled cache retention times

Currently, CacheSack uses 127 predefined cache retention times: 15 minutes, 1.06×15 minutes, $1.06^2 \times 15$ minutes, ..., $1.06^{126} \times 15$ minutes ≈ 16 days; the 128th value is reserved for positive infinity.

These retention times are decided in the following way. We first determine the working range. A retention time less

than 15 minutes means we evict and insert cache blocks in an extremely aggressive way, which would cause serious flash wearout. By policy, any cache block is forced to leave the cache if it stays more than 15 days. Hence we set the modeled working range of retention times as 15 minutes to 15 days. We then decide the number of retention times to model. We tried 127 (6% geometric increase) and 255 (3% geometric increase) retention times, and our experiments showed that 127 retention times gave similar results while reducing RAM usage by half.

6.3 Ghost cache

Since LARC was Colossus Flash Cache's previous admission control, a *ghost cache* was implemented in cache index servers. It is an in-memory lookup table that maps a data's key to the data's last read access time, and LARC uses the information to determine whether to admit the data on miss. CacheSack uses the same ghost cache to obtain inter-arrival times. In addition, to build the metric estimate for `AdmitOnWrite`, we expanded the ghost cache so that we know whether the last access is a write access. To build the metric estimate for `AdmitOnSecondMiss`, we use the ghost cache to record the most two recent access times.

Because the ghost cache is the ground truth for CacheSack, the ghost cache must contain sufficient history. The optimal solution of CacheSack will not be affected as long as the ghost cache TTL, the time since the oldest last access time of the blocks in the ghost cache, is greater than the optimal modeled cache retention time. As a rule of thumb, we provision the size of the ghost cache so that its TTL is at least twice the solved optimal modeled retention time (typically about four hours).

6.4 Buffer cache simulators

A cache miss in Colossus Flash Cache causes a disk read only if it is also a miss in the buffer cache. CacheSack simulates the buffer cache to determine whether the current miss in Colossus Flash Cache is also likely a miss in the buffer cache. In fact, we need *many* simulators: one for each pair of policy-retention time so there are 382 simulators ($3 \times 127 + 1$, the retention time does not affect `NeverAdmit`). Running the simulators is the most computationally intensive component in the CacheSack model. Fortunately, the buffer cache simulator is simple enough and only requires the access history in the past few seconds so it only moderately increases CPU load on the low-QPS servers (5% CPU usage).

6.5 Model training

We use a simple scheme to train the CacheSack model: the model is reset every 5 minutes and is trained based on the lookups in this 5-minute period. We note that a lookup contains the access times of the most recent two accesses and therefore the lookups in a 5-minute period may contain the information of many hours.

The selection of the training duration is a trade-off. Using a larger training duration means the model can be improved by more training data and longer time horizon, while the model can react more quickly to changes in the workload with a shorter duration. We tested several training durations and found that 5-minute one gave most disk read reduction, although we did not find significant differences among all candidates.

6.6 Lessons learned

Automatic cache optimization incentivized user adoption

In deciding whether to use Colossus Flash Cache, users weigh both the likely TCO improvement and the engineering effort required to configure and maintain it. In the past, users had to manually choose the admission policy (using `AdmitOnMiss` or `AdmitOnSecondMiss`) based on knowledge of their workload or by running A/B experiments with the assistance of the Colossus Flash Cache team. For heavy users like Spanner, Colossus Flash Cache had to provide heuristic, hand-tuned admission policies to improve cache performance. Such human tuning and maintenance usually requires effort from both the users and the Colossus Flash Cache team, which can discourage the adoption of Colossus Flash Cache if the expected hardware resource saving does not justify the extra engineering cost.

We found that CacheSack greatly incentivized users to adopt Colossus Flash Cache. The automatic cache provisioning brought by CacheSack requires almost no configuration and maintenance so that it can be set and forgotten. We found that new users were more willing to use Colossus Flash Cache once they knew it would automatically adjust the cache policy based on their workloads.

Some of Colossus Flash Cache's existing users have independently verified that CacheSack applied appropriate admission policies to their workloads, based on the knowledge of their workloads and reporting provided by Colossus Flash Cache. One user experimentally overrode CacheSack with manually optimized policies and found that CacheSack worked as well as manual policy tuning. After CacheSack became the default admission policy in Colossus Flash Cache, we were able to retire the hand-tuned optimization for Spanner, and our existing users did not need to manually adjust the policy anymore.

Experiment infrastructure accelerated feature development

The development of CacheSack was significantly benefited by the experiment infrastructure of Colossus Flash Cache.

The experiment infrastructure allows developers to test new features by using 10% of the cache index servers, and because cache index servers are independent and isolated, any experiment can only cause minor service degradation in the worst case. Before the full deployment, we ran CacheSack as an experiment for a few months and most of the issues were identified and corrected during the experimental phase. In fact, there was no binary rollback caused by CacheSack since the full deployment.

In addition, because each server represents a fraction of the key space, which is permuted randomly, each server is statistically indistinguishable. We can have simultaneous comparisons between CacheSack and the control group to see whether CacheSack works as expected and identify any issues. The experiment infrastructure is extensively used by the developers of Colossus Flash Cache for new features, and the impact of a new feature can be accurately measured before the full deployment.

Model introspectability and maintainability played important roles

We found that the model introspectability played an important role for the adoption of the new cache algorithm. Because any cache algorithm of Colossus Flash Cache will be operated and maintained by developers and site reliability engineers (SREs) after the initial deployment, one requirement of deploying a new cache algorithm is that the model behavior can be fully understood and monitored by the developers and SREs. CacheSack satisfies this requirement as it only assumes that the TTL approximation (Section 5.3) is sufficiently close to the eviction of Colossus Flash Cache, and all model behaviors can be derived from this assumption. Another advantage of a highly introspectable model is that the developers (besides the original designers) of Colossus Flash Cache can easily ensure thorough test coverage, validate software releases, add extend the original functionality of CacheSack without assistance from the original designers. After the deployment of the original CacheSack, it became the foundation of further optimizations for Colossus Flash Cache.

It is also worth mentioning that CacheSack is simple enough to be implemented by limited extensions to the original codebase of Colossus Flash Cache. In particular, the optimization was implemented as a simple greedy algorithm instead of using a generic linear program solver library. This did cost extra time for development, but we decided to do so because it allowed us to minimize the computational overhead and increase system reliability by reducing external dependencies. More importantly, anyone familiar with the ecosystem of Colossus Flash Cache can easily maintain CacheSack or develop new features based on it. The implementation of CacheSack can evolve continuously with Colossus Flash Cache, reducing maintenance burden. Since the completion of the initial deployment, both maintenance and new feature developments have been completely handled by Colossus Flash Cache developers and SREs without the need for involvement from the original designers.

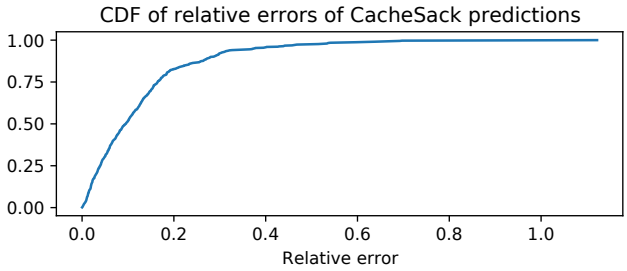

Figure 5: CacheSack disk read rate prediction errors relative to the actual value in production (CDF).

7 Evaluation

7.1 Production evaluation

Model accuracy

There are two LRU approximations in Colossus Flash Cache: Colossus Flash Cache uses Second-Chance-like approach to approximate LRU evictions (Section 4), and CacheSack models an LRU cache as a TTL approximation (Section 5.3). Therefore, it is important to verify that the CacheSack model is a good enough approximation to the actual Colossus Flash Cache. We examined the accuracy of CacheSack in the following way. For each client, the solution to the knapsack problem in Section 5.4 gives the predicted disk reads when using the optimal admission policies. Then Colossus Flash Cache applies the optimal policies in production so we compared the predicted disk reads with the actual disk reads. Figure 5 shows the prediction errors of CacheSack relative to the actual values obtained from the disk servers; 51% of the relative errors are within 10% and 82% of the relative errors are within 20%.

Policy distribution

Figure 6 shows the policy distributions suggested by CacheSack in the selected datacenters of various workloads. We can see that each datacenter has a different workload pattern and CacheSack adaptively decides suitable admission policies based on workloads and cache sizes. Although it would be possible for manual selection of static policies to match each datacenter workload, CacheSack is able to reduce the human toil, response delay, and operational complexity required to maintain these assignments.

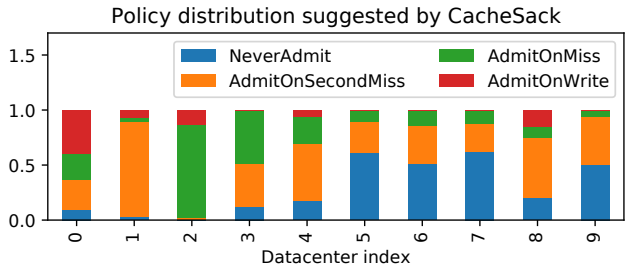

Figure 6: Policy distribution suggested by CacheSack in selected datacenters, demonstrating a variety of workload responses.

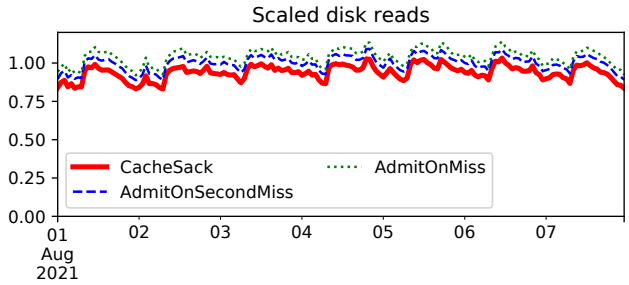

Figure 7: Disk reads of different admission policies in production, divided by the average value for `AdmitOnSecondMiss`.

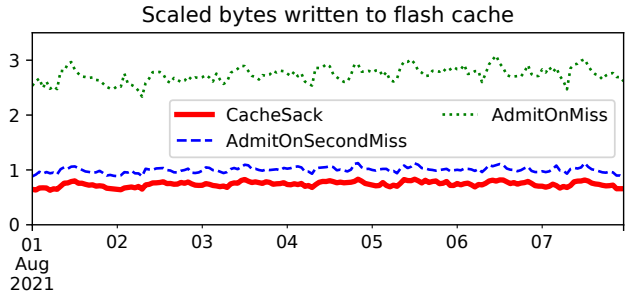

Figure 8: Written bytes of different admission policies in production, divided by the average value for `AdmitOnSecondMiss`.

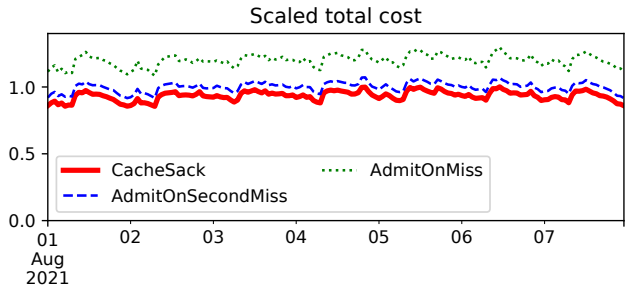

Figure 9: Total cost (a function of disk reads, flash storage and written bytes) of different admission policies in production, divided by the average value for `AdmitOnSecondMiss`.

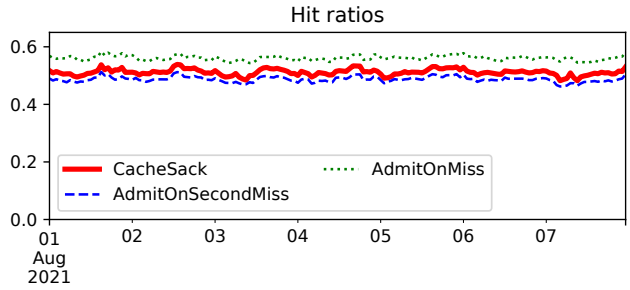

Figure 10: Hit ratios in Colossus Flash Cache of different admission policies in production.

Production experiments

By using the experiment infrastructure of Colossus Flash Cache, we can compare the performance of different cache algorithms in production. Because each cache index server represents a fraction of the key space, the pattern of workload each cache index server receives is statistically indistinguishable. We let 10% of the cache index servers run static `AdmitOnMiss` and another 10% of the cache index servers run `AdmitOnSecondMiss` so that we can compare CacheSack, static `AdmitOnMiss` and static `AdmitOnSecondMiss` simultaneously in production.

From Figure 7 and 8 we see that compared to `AdmitOnSecondMiss`, CacheSack results in fewer disk reads (6% of one week average) and reduces 26% (one week average) written bytes to flash, and Figure 9 shows that CacheSack effectively reduces the total operating cost in production: the cost of disk reads, flash cache writes and flash storage of CacheSack is 93% of `AdmitOnSecondMiss` and is 78% of `AdmitOnMiss` (one week average).

Figure 10 shows that CacheSack has a higher hit ratio than `AdmitOnSecondMiss` but lower than `AdmitOnMiss`. Nevertheless, `AdmitOnMiss` is not the best choice. Figure 7 shows that `AdmitOnMiss` has the worst disk read reduction even though it has the highest hit ratio. Because of the lower-level buffer cache, a higher hit ratio in the flash cache does not necessarily imply fewer disk reads: many major Colossus users optimize their workloads by accessing the same data many times within the first few seconds so that only the first access causes an actual disk read. In this case, `AdmitOnMiss` generates many hits that do not reduce disk reads at all. `AdmitOnSecondMiss` resolves this issue by avoiding a cache insertion if the most recent access time is too recent to expect that the data has left the buffer cache.

7.2 Evaluation by simulations

In addition to production experiments, we also used the *Colossus Flash Cache simulator* to test the performance of CacheSack in a variety of configurations and contexts, such as

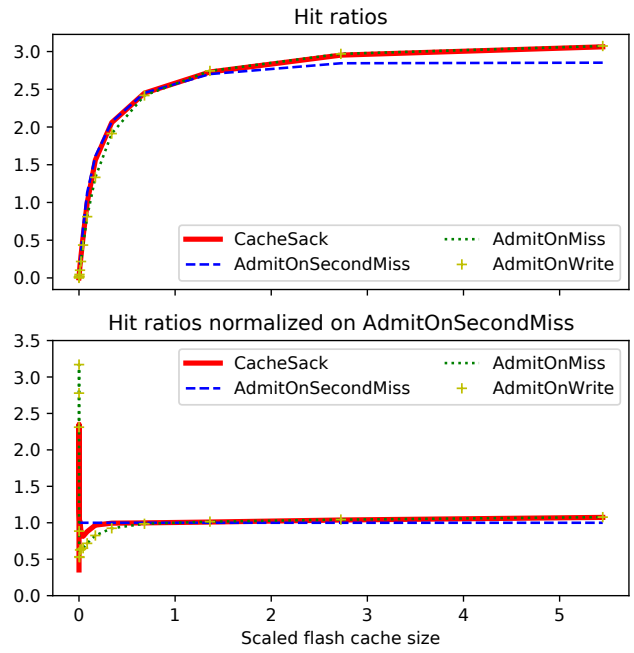

Figure 11: Hit ratios in Colossus Flash Cache of different admission policies in simulation. Above: Original hit ratios. Below: Values relative to `AdmitOnSecondMiss`.

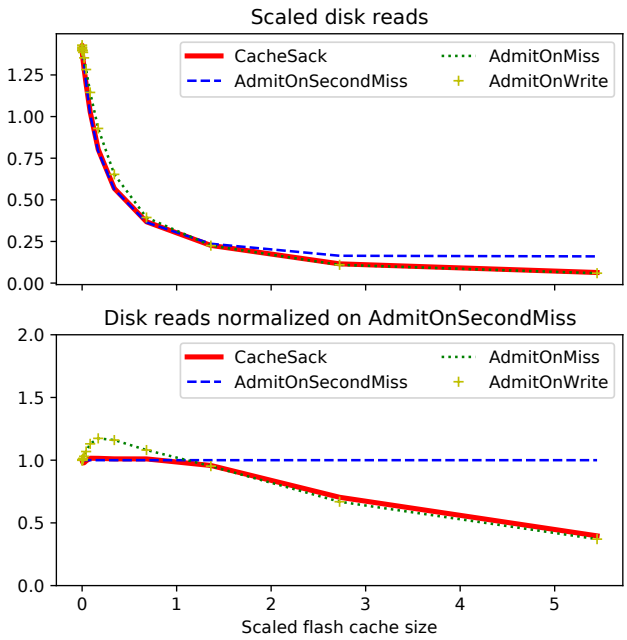

Figure 12: Disk reads of different admission policies in simulation. Above: Constant scaling by dividing the values by the average value for `AdmitOnSecondMiss`. Below: Values relative to `AdmitOnSecondMiss`.

cache size and optimization iteration period. The Colossus Flash Cache simulator is used for multiple purposes including performance-regression testing by Colossus Flash Cache developers and for datacenter resource planning by Colossus Flash Cache clients. The Colossus Flash Cache simulator uses the same production code as Colossus Flash Cache and we use production traces as the input of the simulator.

We first compare the performance of CacheSack, to the static admission policies `AdmitOnMiss`, `AdmitOnSecondMiss` and `AdmitOnWrite` for various cache sizes. We use here a two-day trace from one large (order of million QPS) production cache as a representative. This trace reflects a uniform sample of the data accesses from a large collection of internal production workloads.

Impact of cache size on performance

When the cache size is small, `AdmitOnSecondMiss` has a better performance than `AdmitOnMiss` or `AdmitOnWrite` because single-use keys are excluded. On the other hand, `AdmitOnMiss` and `AdmitOnWrite` will outperform `AdmitOnSecondMiss` for a large cache because the cache benefit of second accesses is gained.

CacheSack learns to use a more conservative policy for a small cache and a more aggressive policy for a large cache. Figure 11 and Figure 12 show that CacheSack can provide a good performance for the entire range of flash cache sizes.

It is also interesting to see the amount of written bytes caused by different admission policies in Figure 14. For `AdmitOnMiss`, `AdmitOnWrite` and `AdmitOnSecondMiss` with excessively small cache, blocks are frequently evicted from and reinserted into the cache, resulting in a very large amount of written bytes, especially for `AdmitOnMiss` and `AdmitOnWrite`. CacheSack, on the other hand, takes into account the cost of written bytes, and therefore only admits the most valuable part of the workload into the cache.

We can also view the total cost (a confidential function of disk reads, flash storage, and written bytes) as a function of cache size. When the cache size is small, disk reads and writes to flash are the largest contributions to cost, while flash storage is the largest cost component for larger cache sizes. Therefore, the total cost is a U-shape curve and we are able to find the optimal cache size that minimizes the total cost. Figure 13 shows that CacheSack gives the lowest total cost for all cache sizes. CacheSack avoids the trade-off and provides robust good behavior over the range of cash sizes.

Optimization frequency

We evaluated the system performance on the choice of different optimization frequencies. Here we test different lengths of training duration from one minute to eight hours, which span a majority of the observed time variation of workloads. Figure 15 shows that the training duration does not significantly impact the performance and all the cost metrics are

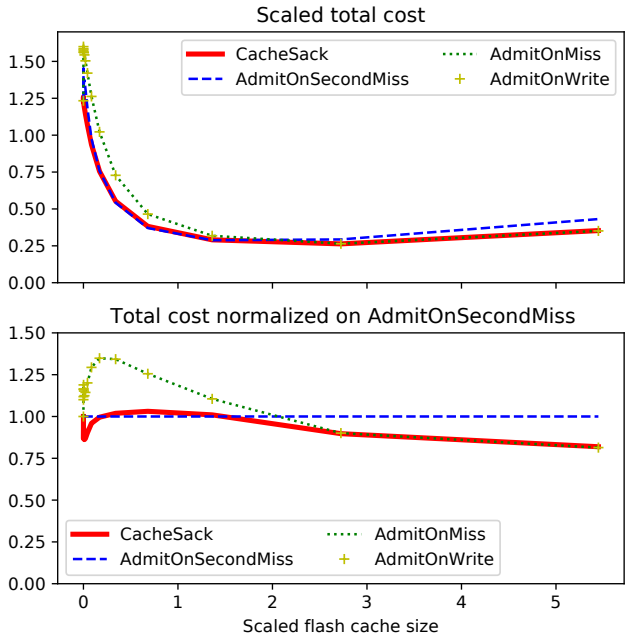

Figure 13: Total cost (a function of disk reads, flash storage and written bytes) of different admission policies in simulation. Above: Constant scaling by dividing the values by the average value for `AdmitOnSecondMiss`. Below: Values relative to `AdmitOnSecondMiss`.

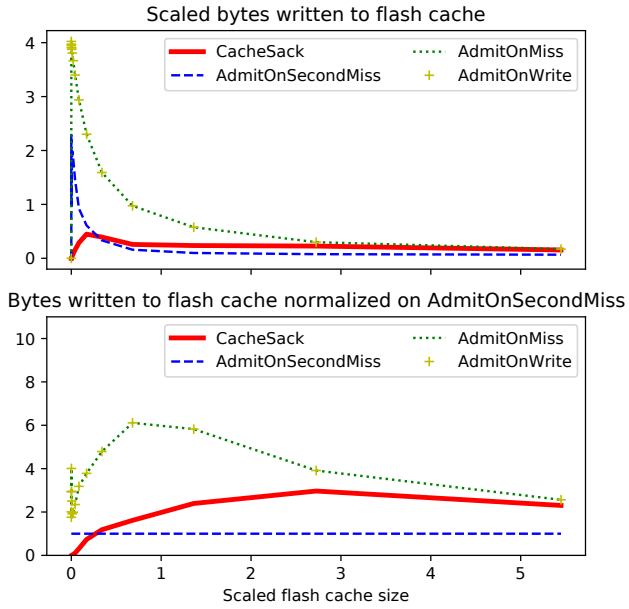

Figure 14: Written bytes of different admission policies in simulation. Above: Constant scaling by dividing the values by the average value for `AdmitOnSecondMiss`. Below: Values relative to `AdmitOnSecondMiss`.

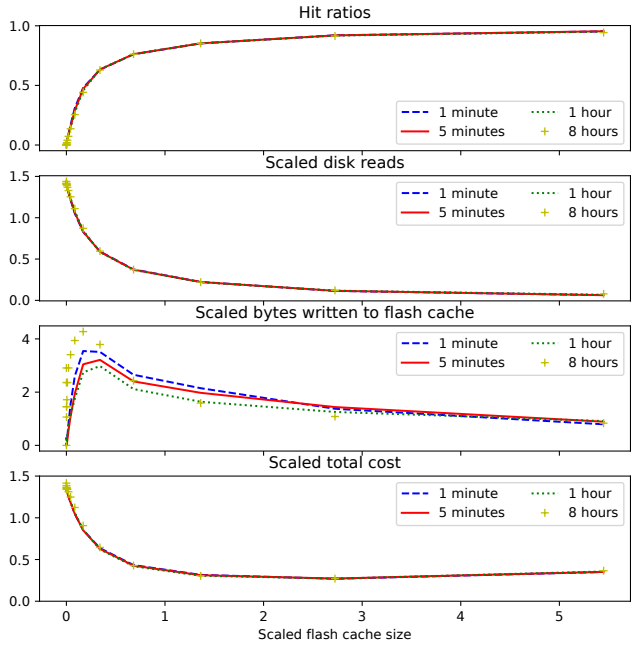

Figure 15: Hit ratios, disk reads, written bytes, and total cost (a function of disk reads, flash storage and written bytes) of Colossus Flash Cache with different training durations. Disk reads, written bytes, and total cost are divided by the average value for the 5-minute training duration.

similar. Because this method is insensitive to this parameter, customized or automated tuning was deemed unneeded, and the entire deployment currently uses a single value.

8 Related work

Production flash cache algorithms

Both Google [1] and Facebook [18] were using FIFO-based evictions in their production flash caches to trade cache performance for managed write amplification. RIPQ [36] is a non-FIFO, advanced flash cache algorithm that brings higher hit ratios while write amplification is well-control. Flashield [16] further improves RIPQ's write amplification by using DRAM as a buffer, and only writes flash-worthy objects into flash, predicted by a lightweight support vector machine classifier. CacheLib [7] resolved Flashield's issue that the TTLs of objects in the DRAM buffer are too short to be effective. CacheLib uses Bloom filters to count the number of accesses per object in the past six hours (similar to TinyLFU [15]), to predict the number of accesses in the future, and use FIFO for eviction. Kangaroo [26] further improves CacheLib's performance for tiny objects. DSS [28] uses predefined rules to classify I/O requests into different priorities, and applies heuristic admission and eviction policies to different priorities. DSS has been implemented in Intel's Cache Acceleration

Software. Amazon's AQUA [3], which is conceptually similar to CacheSack, analyzes workload patterns to place data into the appropriate tier.

CacheSack's high-level idea is similar to Flashield and CacheLib: keep the eviction simple to control write amplification, and use a more sophisticated admission to improve cache performance and flash write endurance. For evictions, Flashield uses the CLOCK [12] approach and CacheSack uses Second Chance [30] to achieve LRU-style evictions. On the admission side, instead of using DRAM as a buffer, CacheSack has no in-memory buffer and expands the metadata table (ghost cache) for a more complete history; the median of the ghost cache TTL is 20 hours, which is several times longer than the information used by Flashield and CacheLib. With a more complete history, CacheSack is able to build a more sophisticated model for admission. CacheSack also considers the two major costs of operating flash caches, disk reads and flash wearout, as a whole, and minimizes the overall operation cost.

CacheSack also utilizes the advantage that the categories are well-defined in the database systems served by Colossus Flash Cache. Classifying unstructured data is usually a difficult problem in machine learning. For Google's database systems, the classification is naturally given, and the categories often hint their cacheability.

Admission algorithms

LARC [21] was previously used by Colossus Flash Cache as the default admission policy. LARC is designed for flash caches, and reduces write rate by inserting an object into the cache only when it is read a second time, based on the observation that most objects are read only once. Thus, inserting only the objects that are read a second time into the cache significantly reduces the write rate and the cache pollution. This strategy is particularly useful when a significant portion of the traffic is accessed only once, for example, Tencent's photo traffic [41], and AliCloud [24]. However, LARC loses all the first cache hits and becomes undesirable for long-tail accesses like Facebook's cache for social network photos. In the past, Colossus Flash Cache disabled LARC for workloads in which LARC underperformed. Selective admissions like TinyLFU [15] (non-window version) and HEC [42] that sacrifice the first few hits to determine the cacheability of data likely have the same issue.

TinyLFU [15] works by comparing the expected hit ratio of a newly accessed object against that of the object that would be evicted next from the cache, inserting the new object if its likely hit ratio is higher. Any eviction policy can be used to select the eviction victim (LRU is typical). TinyLFU predicts hit ratios for the objects using approximate counting (Bloom filters) of access frequency. TinyLFU also needs some extra structures to work properly: *Doorkeeper* is used to filter one-accessed blocks (the same use of LARC's ghost cache), and a DRAM buffer cache in front of TinyLFU (W-TinyLFU). All these structures require extra parameter tuning, which does not best fit the needs of Colossus Flash Cache as a general-purpose cache. mARC [32] uses ARC [27] as the eviction policy and dynamically determines whether to admit data on the first miss (naive ARC) or second miss (LARC). UBC [31] proposes a low-overhead mechanism to partition shared on-chip cache.

Eviction algorithms

There are also extensive studies on advanced eviction algorithms. Beckmann and Sanchez's method [5] evicts a block based on the block's economic value added. Instead of LRU or LFU that require specific data structures, Hyperbolic Caching [9] evicts a block based on a time-decay (hyperbolic) value function and uses a sampling technique to resolve the issue of the data structure requirement. Similarly, LHD [4] evicts the block of the lowest hit density, the number of hits per cache byte-second, and also applies a sampling technique to overcome the data structure issue. Hawkeye [22] assumes that the recent history can predict the near future and hence one can train a predictor learned from Belady's OPT [6] running on the recent traces. [40] considers an ensemble of candidates, which can be a set of existing algorithms, or the same algorithm with different parameters, runs scaled-down simulations on each candidate and periodically adopts the most performant one.

Machine learning algorithms

With the recent rapid development of machine learning (ML), there are also a few papers that adopt ML techniques to enhance cache performance. LFO [8] and LRB [35] use ML models to learn Belady's OPT [6], and apply the ML models to CDN (Content Delivery Networks) caches. Parrot [25] also use ML to learn Belady's OPT from history, but uses modern deep learning architectures like Transformer [38] and BiDAF [33]. [41] utilizes a concept similar to LARC [21] that the majority of traffic is accessed just once, and uses ML models to predict whether data is worth inserting into the flash cache. The algorithm showed a large flash write reduction in Tencent's photo cache system as well as the improvement of hit ratios and latency. LeCaR [39] uses an ML approach to adaptively decide the better policy between LRU and LFU at eviction time. Zhou and Maas [43] model the inter-arrival times of a block as a log-normal distribution and learn the parameters from traces; then the evictions are executed in the manner of Belady's OPT: the block with lowest probability to get the next access in the near future will be evicted.

9 Conclusions

In this paper, we introduce CacheSack, an admission policy optimization for Google's datacenter flash caches. CacheSack provides an efficient estimation for the performance metrics of an LRU-style cache under various configuration options. We use a knapsack approach to identify the optimal admission polices to minimize the total cache-operating cost. We share the experience of deploying CacheSack in Colossus Flash Cache, the general-purpose flash cache serving Colossus, which has since become the default admission policy. CacheSack requires less manual configuration than the previously used cache admission algorithm (LARC), significantly reduces disk reads and bytes written to flash, and improves TCO by 6.5% compared to LARC.

Acknowledgements

The authors would like to thank Cory Casper, Junaid Khalid, Martin Maas, and Richard McDougall for their assistance in various stages of the design and implementation of this algorithm. We would also like to thank Dan Gibson, Larry Greenfield, Aaron Laursen, Milo Martin, Damodharan Rajalingam, and John Wilkes, as well as the anonymous reviewers and our conference shepherd, for their reviews and suggestions that improved this manuscript.

A Mathematical model of CacheSack

A.1 Model assumption

CacheSack models the cache as using LRU evictions. Colossus Flash Cache considers data for caching to be immutable after being written, i.e. the first access is a write, and subsequent ones are reads; mutability is handled by higher layers in the system. The CacheSack model does not need the immutability assumption, but we keep it to align with the actual system; the model can be easily modified for the mutable case.

A.2 Metric estimation of an LRU cache

We begin with `AdmitOnMiss`. For a given block b, let t_1, t_2, t_3, ..., t_n be the read access times, and $t_0 = -\infty$ for convenience. Therefore, the inter-arrival times are $d_i = t_i - t_{i-1}$ and $d_1 = t_1 - t_0 = \infty$. Assume that D is the modeled cache retention time; that is, D is the maximum duration that a block stays in the LRU cache without any intervening accesses. We can classify the inter-arrival times as follows:

- $d_i \le D$: A cache hit because the access arrives before the block leaves the cache. The block is moved to the head of the queue because of the LRU eviction.
- $d_i > D$: A cache miss because the access arrives after the block leaves the cache. `AdmitOnMiss` inserts the block into the cache on miss, causing a write to the cache.
- For a flash cache miss, we update the *buffer cache simulator* to see whether it is also a cache miss in the buffer cache. If so, the access is a disk read.

We can then write disk reads $S_b^{\texttt{AOM}}(D)$, cache byte-time usage[2] $U_b^{\texttt{AOM}}(D)$ and bytes written to cache $W_b^{\texttt{AOM}}(D)$ as functions of D:

$$B_b^{\texttt{AOM}}(D,i) = \begin{cases} 1, & \text{Buffer Cache Hit at } t_i, \text{ using } \texttt{AOM}. \\ 0, & \text{Buffer Cache Miss at } t_i, \text{ using } \texttt{AOM}. \end{cases}$$

$$S_b^{\texttt{AOM}}(D) = |\{i : d_i > D, B_b^{\texttt{AOM}}(D,i) = 0\}|,$$

$$U_b^{\texttt{AOM}}(D) = \text{Size}(b) \times \sum_i \min(d_i, D),$$

$$W_b^{\texttt{AOM}}(D) = \text{Size}(b) \times |\{i : d_i > D\}|.$$

Similarly, the metrics for a category $\mathcal{C}$ is the sum of the metrics for all blocks in $\mathcal{C}$:

$$S_{\mathcal{C}}^{\texttt{AOM}}(D) = \sum_{b \in \mathcal{C}} S_b^{\texttt{AOM}}(D),$$

$$U_{\mathcal{C}}^{\texttt{AOM}}(D) = \sum_{b \in \mathcal{C}} U_b^{\texttt{AOM}}(D),$$

$$W_{\mathcal{C}}^{\texttt{AOM}}(D) = \sum_{b \in \mathcal{C}} W_b^{\texttt{AOM}}(D).$$

The only difference between `AdmitOnWrite` and `AdmitOnMiss` is that `AdmitOnWrite` also takes into account write accesses. Therefore, for `AdmitOnWrite`, we let t_1 be the write access time, t_2 be the first read access time, t_3 be the second read access time and so on. Then we can similarly define $S_{\mathcal{C}}^{\texttt{AOW}}(D)$, $U_{\mathcal{C}}^{\texttt{AOW}}(D)$ and $W_{\mathcal{C}}^{\texttt{AOW}}(D)$.

For `AdmitOnSecondMiss`, a block is admitted at the second miss (read access). In addition, to prevent the cache from inserting a cold block, we require that when inserting a block, its last read access time must be not older than the oldest last access time of the blocks in the cache. Mathematically, a block is inserted at t_{i-1} (if not already in the cache) only if $d_{i-1} = t_{i-1} - t_{i-2} \le D$. Therefore, the condition that a block is in the cache at t_{i-1}, either because it is already in the cache or it is inserted, is $d_{i-1} \le D$, and so an access at t_i is a cache hit if and only if $d_{i-1} \le D$ and $d_i \le D$:

$$B_b^{\texttt{AOSM}}(D,i) = \begin{cases} 1, & \text{Buffer Cache Hit at } t_i, \text{ using } \texttt{AOSM}. \\ 0, & \text{Buffer Cache Miss at } t_i, \text{ using } \texttt{AOSM}. \end{cases}$$

$$S_b^{\texttt{AOSM}}(D) = |\{i : \max(d_{i-1}, d_i) > D, B_b^{\texttt{AOSM}}(D,i) = 0\}|.$$

For $U_b^{\texttt{AOSM}}(D)$, the access at t_i contributes cache usage if either it is a cache hit, $\max(d_i, d_{i-1}) \le D$, with residence time d_i, or

[2] Bytes of occupied cache multiplied by seconds of residence time in cache. The same concept is also used in LHD [4].

a block insertion, $d_i \le D < d_{i-1}$, with residence time D:

$$U_b^{\texttt{AOSM}}(D) = \text{Size}(b) \times \sum_i \Big(d_i \times \mathbf{1}_{\{\max(d_i, d_{i-1}) \le D\}} + D \times \mathbf{1}_{\{d_i \le D < d_{i-1}\}} \Big).$$

$W_b^{\texttt{AOSM}}(D)$ is the block size times the number of insertions:

$$W_b^{\texttt{AOSM}}(D) = \text{Size}(b) \times |\{i : d_i \le D < d_{i-1}\}|.$$

Of course, $S_{\mathcal{C}}^{\texttt{AOSM}}(D)$, $U_{\mathcal{C}}^{\texttt{AOSM}}(D)$ and $W_{\mathcal{C}}^{\texttt{AOSM}}(D)$ can be defined similarly.

Because `NeverAdmit` does not insert any blocks at all, $U_{\mathcal{C}}^{\texttt{NA}}(D) = 0$, $W_{\mathcal{C}}^{\texttt{NA}}(D) = 0$ and $S_{\mathcal{C}}^{\texttt{NA}}(D)$ is the number of buffer cache misses because of the accesses to the blocks in $\mathcal{C}$:

$$B_b^{\texttt{NA}}(D,i) = \begin{cases} 1, & \text{Buffer Cache Hit at } t_i, \text{ using } \texttt{NA}. \\ 0, & \text{Buffer Cache Miss at } t_i, \text{ using } \texttt{NA}. \end{cases}$$

$$S_b^{\texttt{NA}}(D) = |\{i : B_b^{\texttt{NA}}(D,i) = 0\}|,$$

$$S_{\mathcal{C}}^{\texttt{NA}}(D) = \sum_{b \in \mathcal{C}} S_b^{\texttt{NA}}(D).$$

A.3 Linear program

We minimize the total cost by formulating a linear program. The cost function is the sum of the cost of disk reads and the cost of written bytes:

$$V_{\mathcal{C}}^{p}(D) = \text{Cost of } S_{\mathcal{C}}^{p}(D) + \text{Cost of } W_{\mathcal{C}}^{p}(D),$$

for $p \in \{\texttt{AOM}, \texttt{AOW}, \texttt{AOSM}, \texttt{NA}\}$ and a given category $\mathcal{C}$.

A category can receive *fractional* admission policies. For example, CacheSack may decide that it is optimal to apply `AdmitOnMiss`, `AdmitOnSecondMiss`, `AdmitOnWrite` and `NeverAdmit` are applied to 30%, 20%, 10% and 40% of blocks in $\mathcal{C}$, respectively. Then we can formulate a linear program that finds optimal policy fractions $\{\alpha_{\mathcal{C}}^{\texttt{AOM}}, \alpha_{\mathcal{C}}^{\texttt{AOW}}, \alpha_{\mathcal{C}}^{\texttt{AOSM}}, \alpha_{\mathcal{C}}^{\texttt{NA}}\}$ to minimize the overall cost:

$$\min_{\alpha_{\mathcal{C}}^{\texttt{AOM}}, \alpha_{\mathcal{C}}^{\texttt{AOW}}, \alpha_{\mathcal{C}}^{\texttt{AOSM}}, \alpha_{\mathcal{C}}^{\texttt{NA}}} \sum_{\mathcal{C}} \Big(\alpha_{\mathcal{C}}^{\texttt{AOM}} V_{\mathcal{C}}^{\texttt{AOM}}(D) + \alpha_{\mathcal{C}}^{\texttt{AOSM}} V_{\mathcal{C}}^{\texttt{AOSM}}(D) + \alpha_{\mathcal{C}}^{\texttt{AOW}} V_{\mathcal{C}}^{\texttt{A)W}}(D) + \alpha_{\mathcal{C}}^{\texttt{NA}} V_{\mathcal{C}}^{\texttt{NA}}(D) \Big), \tag{1}$$

subject to the capacity constraint that the cache usage should not exceed the given cache capacity U_{total}:

$$0 \le \alpha_{\mathcal{C}}^{\texttt{AOM}}, \alpha_{\mathcal{C}}^{\texttt{AOW}}, \alpha_{\mathcal{C}}^{\texttt{AOSM}}, \alpha_{\mathcal{C}}^{\texttt{NA}} \le 1,$$

$$\alpha_{\mathcal{C}}^{\texttt{AOM}} + \alpha_{\mathcal{C}}^{\texttt{AOW}} + \alpha_{\mathcal{C}}^{\texttt{AOSM}} + \alpha_{\mathcal{C}}^{\texttt{NA}} = 1,$$

$$\sum_{\mathcal{C}} \Big(\alpha_{\mathcal{C}}^{\texttt{AOM}} U_{\mathcal{C}}^{\texttt{AOM}}(D) + \alpha_{\mathcal{C}}^{\texttt{AOSM}} U_{\mathcal{C}}^{\texttt{AOSM}}(D) + \alpha_{\mathcal{C}}^{\texttt{AOW}} U_{\mathcal{C}}^{\texttt{AOW}}(D) + \alpha_{\mathcal{C}}^{\texttt{NA}} U_{\mathcal{C}}^{\texttt{NA}}(D) \Big) \le U_{\text{total}}.$$

We note that if the LRU cache is perfectly modeled by the approach in Section A.2, the resulting cache retention time of the LRU cache is exactly D after applying the optimal policy fractions.

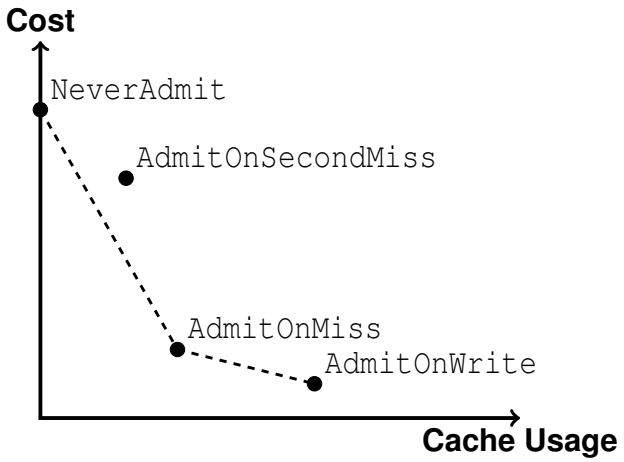

Figure 16: Example of Andrew's monotone chain convex hull algorithm applied to the admission policies.

A.4 Greedy algorithm

Although the linear program (1) can be solved by a standard solver, we are able to solve it by a greedy algorithm with a simple transformation. It is especially beneficial for the production deployment because of the low-overhead and stability of the greedy algorithm, compared to a generic solver. We first note that the difference between the above linear program and a fractional knapsack problem [14] is that for each category, we need to decide *coupled* four fractions (three degrees of freedom), instead of two fractions (one degree of freedom) in a fractional knapsack problem. Thus, the greedy algorithm in [14] can not be directly applied. However, we can use Andrew's lower convex hull algorithm [2] to decouple the dependency.

For a given category $\mathcal{C}$, the lower convex hull formed by the points $\{(U_{\mathcal{C}}^{p}, V_{\mathcal{C}}^{p}), p \in \{\texttt{AOM}, \texttt{AOW}, \texttt{AOSM}, \texttt{NA}\}\}$, is the lowest cost of $\mathcal{C}$ that can be generated among all convex combinations of the policies. For example, Figure 16 is the lower convex hull constructed by the given admission policies by using Andrew's algorithm. Let $F_{\mathcal{C}}(u)$ denote the lower convex hull formed above, as a mapping from cache usage u to the corresponding cost, for each category $\mathcal{C}$. By dropping any line segments with non-negative slopes, all $F_{\mathcal{C}}$ are strictly decreasing, piecewise linear functions. Then we can transform the linear program to a convex optimization problem:

$$\min_{u_{\mathcal{C}} \ge 0} \sum_{\mathcal{C}} F_{\mathcal{C}}(u_{\mathcal{C}}), \quad \sum_{\mathcal{C}} u_{\mathcal{C}} \le U_{\text{total}}.$$

We can then solve the above convex optimization problem by the steepest descent method (a greedy algorithm). We initialize $u_{\mathcal{C}} = 0$ for all $\mathcal{C}$ and iteratively decide each $u_{\mathcal{C}}$ in the following way. We first choose the line segment with the most negative slope among all line segments of $F_{\mathcal{C}}$ and change the value of the corresponding $u_{\mathcal{C}}$. In the same fashion, we

then choose the line segment with second most negative slope and change the value of the corresponding u_C, then the third most negative slope, and so on, until the sum of u_C reaches U_{total}.

Because we allow fractional policies, the category corresponding to the last chosen line segment generally has the optimal policy as a convex combination of two of $\{\texttt{AOM}, \texttt{AOW}, \texttt{AOSM}, \texttt{NA}\}$, and the optimal policy of any other category must be exactly one of $\{\texttt{AOM}, \texttt{AOW}, \texttt{AOSM}, \texttt{NA}\}$.

A.5 Optimization over modeled cache retention times

The linear program (1) is to find the optimal policy fractions for a *given* modeled cache retention time D, which can not be known *a priori*. Thus, we need to solve the same optimization problem for *all* possible D:

$$\min_{D>0} \min_{\alpha_C^{\texttt{AOM}}, \alpha_C^{\texttt{AOW}}, \alpha_C^{\texttt{AOSM}}, \alpha_C^{\texttt{NA}}} \sum_C \Big(\alpha_C^{\texttt{AOM}} V_C^{\texttt{AOM}}(D) + \alpha_C^{\texttt{AOSM}} V_C^{\texttt{AOSM}}(D) + \alpha_C^{\texttt{AOW}} V_C^{\texttt{AOW}}(D) + \alpha_C^{\texttt{NA}} V_C^{\texttt{NA}}(D) \Big),$$

subject to the same capacity constraint.

To do this, we can use a standard scalar-variable optimization approach like Brent's method [10] for $0 < D \leq \mathbf{D}$, where $\mathbf{D}$ is a suitable upper bound. A brute-force approach may be even more practical for implementation: we simply solve the optimization problem for a set of reasonable retention times: $0 < D_1 < D_2 < \cdots < D_m = \mathbf{D}$.

References

[1] Christoph Albrecht, Arif Merchant, Murray Stokely, Muhammad Waliji, François Labelle, Nate Coehlo, Xudong Shi, and C. Eric Schrock. Janus: Optimal Flash Provisioning for Cloud Storage Workloads. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 91–102. USENIX Association, June 2013.

[2] A.M. Andrew. Another efficient algorithm for convex hulls in two dimensions. *Information Processing Letters*, 9(5):216–219, 1979.

[3] Jeff Barr. AQUA (Advanced Query Accelerator) – A Speed Boost for Your Amazon Redshift Queries. https://aws.amazon.com/blogs/aws/new-aqua-advanced-query-accelerator-for-amazon-redshift/, April 2021.

[4] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. LHD: Improving Cache Hit Rate by Maximizing Hit Density. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 389–403. USENIX Association, April 2018.

[5] Nathan Beckmann and Daniel Sanchez. Maximizing Cache Performance Under Uncertainty. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 109–120, 2017.

[6] L. A. Belady. A Study of Replacement Algorithms for a Virtual-Storage Computer. *IBM Syst. J.*, 5(2):78–101, June 1966.

[7] Benjamin Berg, Daniel S. Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, and Gregory R. Ganger. The CacheLib Caching Engine: Design and Experiences at Scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 753–768. USENIX Association, November 2020.

[8] Daniel S. Berger. Towards Lightweight and Robust Machine Learning for CDN Caching. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*, HotNets '18, page 134–140. Association for Computing Machinery, 2018.

[9] Aaron Blankstein, Siddhartha Sen, and Michael J. Freedman. Hyperbolic Caching: Flexible Caching for Web Applications. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 499–511. USENIX Association, July 2017.

[10] R.P. Brent. *Algorithms for minimization without derivatives*. Prentice-Hall, 1973.

[11] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.*, 26(2), June 2008.

[12] F.J. Corbató. *A Paging Experiment with the Multics System*. Massachusetts Institute of Technology, 1968.

[13] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally-Distributed Database. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 261–264. USENIX Association, October 2012.

[14] George B. Dantzig. Discrete-Variable Extremum Problems. *Operations Research*, 5(2):266–288, 1957.

[15] Gil Einziger, Roy Friedman, and Ben Manes. TinyLFU: A Highly Efficient Cache Admission Policy. *ACM Trans. Storage*, 13(4), November 2017.

[16] Assaf Eisenman, Asaf Cidon, Evgenya Pergament, Or Haimovich, Ryan Stutsman, Mohammad Alizadeh, and Sachin Katti. Flashield: a Hybrid Key-value Cache that Controls Flash Write Amplification. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 65–78. USENIX Association, February 2019.

[17] Ronald Fagin. Asymptotic miss ratios over independent references. *Journal of Computer and System Sciences*, 14(2):222–250, 1977.

[18] Alex Gartrell. McDipper: A key-value cache for Flash storage. https://engineering.fb.com/2013/03/05/web/mcdipper-a-key-value-cache-for-flash-storage/, March 2013.

[19] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 20–43, 2003.

[20] Dean Hildebrand and Denis Serenyi. Colossus under the hood: a peek into Google's scalable storage system. https://cloud.google.com/blog/products/storage-data-transfer/a-peek-behind-colossus-googles-file-system, April 2021.

[21] Sai Huang, Qingsong Wei, Dan Feng, Jianxi Chen, and Cheng Chen. Improving Flash-Based Disk Cache with Lazy Adaptive Replacement. *ACM Trans. Storage*, 12(2), February 2016.

[22] Akanksha Jain and Calvin Lin. Back to the Future: Leveraging Belady's Algorithm for Improved Cache Replacement. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, page 78–89. IEEE Press, 2016.

[23] Bo Jiang, Philippe Nain, and Don Towsley. On the Convergence of the TTL Approximation for an LRU Cache under Independent Stationary Request Processes. *ACM Trans. Model. Perform. Eval. Comput. Syst.*, 3(4), September 2018.

[24] Jinhong Li, Qiuping Wang, Patrick P. C. Lee, and Chao Shi. An In-Depth Analysis of Cloud Block Storage Workloads in Large-Scale Production. In *2020 IEEE International Symposium on Workload Characterization (IISWC)*, pages 37–47, 2020.

[25] Evan Zheran Liu, Milad Hashemi, Kevin Swersky, Parthasarathy Ranganathan, and Junwhan Ahn. An Imitation Learning Approach for Cache Replacement. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, pages 6237–6247. PMLR, 2020.

[26] Sara McAllister, Benjamin Berg, Julian Tutuncu-Macias, Juncheng Yang, Sathya Gunasekar, Jimmy Lu, Daniel S. Berger, Nathan Beckmann, and Gregory R. Ganger. Kangaroo: Caching Billions of Tiny Objects on Flash. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 243–262. Association for Computing Machinery, 2021.

[27] Nimrod Megiddo and Dharmendra S. Modha. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *2nd USENIX Conference on File and Storage Technologies (FAST 03)*. USENIX Association, March 2003.

[28] Michael Mesnier, Feng Chen, Tian Luo, and Jason B. Akers. Differentiated storage services. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, page 57–70. Association for Computing Machinery, 2011.

[29] Changwoo Min, Kangnyeon Kim, Hyunjin Cho, Sang-Won Lee, and Young Ik Eom. SFS: Random Write Considered Harmful in Solid State Drives. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, page 12. USENIX Association, 2012.

[30] Pancham Pancham, Deepak Chaudhary, and Ruchin Gupta. Comparison of Cache Page Replacement Techniques to Enhance Cache Memory Performance. *International Journal of Computer Applications*, 98:27–33, 07 2014.

[31] Moinuddin K. Qureshi and Yale N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pages 423–432, 2006.

[32] Ricardo Santana, Steven Lyons, Ricardo Koller, Raju Rangaswami, and Jason Liu. To ARC or Not to ARC. In *7th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 15)*. USENIX Association, July 2015.

[33] Min Joon Seo, Aniruddha Kembhavi, Ali Farhadi, and Hannaneh Hajishirzi. Bidirectional Attention Flow for Machine Comprehension. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[34] Jeff Shute, Radek Vingralek, Bart Samwel, Ben Handy, Chad Whipkey, Eric Rollins, Mircea Oancea, Kyle Littlefield, David Menestrina, Stephan Ellner, John Cieslewicz, Ian Rae, Traian Stancescu, and Himani Apte. F1: A Distributed SQL Database That Scales. In *VLDB*, 2013.

[35] Zhenyu Song, Daniel S. Berger, Kai Li, and Wyatt Lloyd. Learning relaxed belady for content distribution network caching. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 529–544. USENIX Association, February 2020.

[36] Linpeng Tang, Qi Huang, Wyatt Lloyd, Sanjeev Kumar, and Kai Li. RIPQ: Advanced Photo Caching on Flash for Facebook. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 373–386. USENIX Association, February 2015.

[37] Rajesh Thallam. BigQuery explained: An overview of BigQuery's architecture. https://cloud.google.com/blog/products/data-analytics/new-blog-series-bigquery-explained-overview, September 2020.

[38] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is All you Need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[39] Giuseppe Vietri, Liana V. Rodriguez, Wendy A. Martinez, Steven Lyons, Jason Liu, Raju Rangaswami, Ming Zhao, and Giri Narasimhan. Driving Cache Replacement with ML-Based LeCaR. In *Proceedings of the 10th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'18, page 3. USENIX Association, 2018.

[40] Carl Waldspurger, Trausti Saemundsson, Irfan Ahmad, and Nohhyun Park. Cache Modeling and Optimization using Miniature Simulations. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 487–498. USENIX Association, July 2017.

[41] Hua Wang, Xinbo Yi, Ping Huang, Bin Cheng, and Ke Zhou. Efficient SSD Caching by Avoiding Unnecessary Writes Using Machine Learning. In *Proceedings of the 47th International Conference on Parallel Processing*, ICPP 2018. Association for Computing Machinery, 2018.

[42] Jingpei Yang, Ned Plasson, Greg Gillis, Nisha Talagala, Swaminathan Sundararaman, and Robert Wood. HEC: Improving Endurance of High Performance Flash-Based Cache Devices. In *Proceedings of the 6th International Systems and Storage Conference*, SYSTOR '13. Association for Computing Machinery, 2013.

[43] Giulio Zhou and Martin Maas. Learning on distributed traces for data center storage systems. In A. Smola, A. Dimakis, and I. Stoica, editors, *Proceedings of Machine Learning and Systems*, volume 3, pages 350–364, 2021.

[44] Huapeng Zhou, Linpeng Tang, Qi Huang, and Wyatt Lloyd. The Evolution of Advanced Caching in the Facebook CDN. https://research.fb.com/blog/2016/04/the-evolution-of-advanced-caching-in-the-facebook-cdn/, April 2016.