PaperCache: In-Memory Caching with Dynamic Eviction Policies

Kia Shakiba and Michael Stumm
University of Toronto

Abstract

In-memory caches play a critical role in storage environments by reducing data access latencies and loads on backend data stores. A cache's eviction policy significantly impacts its attained miss ratio, and recent modeling techniques allow for efficient evaluation of different eviction policies at runtime. However, modern in-memory caches lack the ability to switch between eviction policies at runtime, except for Redis that can only switch between LRU and LFU. We present PaperCache, an in-memory cache capable of switching between multiple different eviction policies at runtime. Our evaluation shows that immediately after an eviction policy switch, PaperCache's behavior closely mirrors that of a cache implementing the target policy exactly (with a miss ratio typically within 1%) for a short period of time, after which PaperCache's behavior is fully inline with an exact policy implementation. Further, PaperCache is able to periodically and automatically switch to the policy exhibiting the lowest miss ratio, reducing the overall miss ratio by up to 48.5%.

CCS Concepts

• **Information systems → Cloud based storage**; *Storage management*.

Keywords

In-memory caching, dynamic eviction policy switching

ACM Reference Format:
Kia Shakiba and Michael Stumm. 2025. PaperCache: In-Memory Caching with Dynamic Eviction Policies. In *The 17th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage '25), July 10–11, 2025, Boston, MA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3736548.3737836

1 Introduction

In-memory caches are pervasive in large-scale storage environments. They play an important role in reducing data access latency and the load on backend data stores by serving data directly from DRAM [1–15]. They store frequently accessed, "hot" data in the form of key-value pairs (referred to as *objects*). Although DRAM offers faster data access, it has a higher operational cost and is thus often significantly smaller than the backend data stores. In-memory caches typically only hold a small subset of the data in the backend stores; they use an *eviction policy* to select data for removal from the cache to make room for new data to be inserted.

A cache's eviction policy can significantly affect its performance, typically measured as the cache's *miss ratio* (i.e., the ratio of the number of accesses to data not found in the cache to the total number of accesses) [8]. The *least recently used* (**LRU**) eviction policy is the most widely deployed policy, and is used as the default (and in most cases only) policy in popular in-memory caches, such as Redis [16] or Memcached [17]. However, many alternative eviction policies exist, such as *least frequently used* (**LFU**), *first-in-first-out* (**FIFO**), *most recently used* (**MRU**), 2Q [18], *least recently/frequently used* (**LRFU**) [19], ARC [20], S3-FIFO [10], LHD [11], or SIEVE [21], which have been shown to outperform LRU for specific workloads.

Recently proposed cache performance modeling techniques, such as Kosmo [8] and MiniSim [22], are able to accurately determine a cache's optimal eviction policy based on its workload and allocated size. These algorithms generate *miss ratio curves* (**MRCs**) which plot a cache's miss ratio as a function of its allocated size. Unfortunately, most modern caches do not support multiple eviction policies, and those that do, such as Redis [16], make compromises on policy accuracy to reduce management overhead (§2.1).[1]

This paper introduces **PaperCache**, a novel cache design that supports dynamic switching between any eviction policy during runtime. PaperCache uses a unique eviction policy approximation technique with a *MiniStack* to temporarily approximate the behavior of an eviction policy while switching between policies and servicing new accesses. We evaluate the performance of PaperCache and the accuracy of MiniStacks

[1]CacheLib [23] also supports multiple eviction policies, including arbitrary user-definable policies. However, it does not support the dynamic switching between them at runtime which we show is important in §2.3.

using publicly-available real-world cache access traces from Twitter [3], Cloudphysics [24], IBM [25], Tencent CBS [26], Alibaba [27], and Wikipedia [28]. In §4, we show that PaperCache can switch between eviction policies instantaneously while continuing to service new accesses and achieves a miss ratio within 1% of exact implementations of its eviction policies for a short period of time. We demonstrate that PaperCache's ability to periodically switch to its most performant eviction policy at runtime can reduce the miss ratio by between 8.2% and 48.5% when compared to statically configured eviction policies.

Contributions. We make the following contributions:

- We demonstrate that Redis' eviction policy approximation technique leads to inaccurate policy behavior after switching policies during runtime (§2.1).
- We demonstrate that the optimal eviction policy for a cache can change over time (§2.3).
- We introduce and evaluate PaperCache, an in-memory cache that supports the efficient dynamic switching between any eviction policy in real-time (§3).

Limitations. Our work has the following limitations:

- Our analysis is based on publicly-available workloads and our findings may not apply to all caching environments.
- PaperCache has higher metadata memory overhead than, say, Redis (10.8% and 70% higher when storing between 100,000 and 1,000,000 objects, respectively, when configured with 8 eviction policies).

2 Background and motivation

In this section, we describe relevant prior work and the motivation for PaperCache. We first describe how Redis implements its LRU and LFU eviction policies, and how it switches between them (§2.1). Next, we describe SHARDS, a sampling technique which is used by PaperCache to reduce the overhead of storing eviction policy metadata (§2.2). Finally, we show that the optimal eviction policy for a cache may change over time as motivation for PaperCache's dynamic eviction policy switching capabilities (§2.3).

2.1 Multi-eviction policy support in Redis

Popular open-source in-memory caches are rather limited in the number of eviction policies they support. Memcached [17] only supports LRU, while other caches, such as CacheLib [23], support multiple eviction policies but cannot switch between them at runtime. Redis supports LRU and LFU and can switch between them at runtime [16]. In Redis, switching the eviction policy is a *passive* configuration change (i.e., it does not trigger any action by an idle cache). Redis employs several performance optimizations, which makes its eviction policies only approximations of LRU and

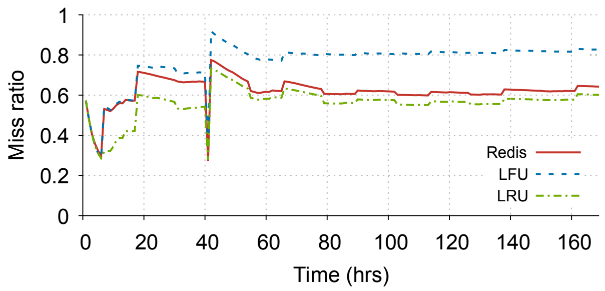

Figure 1: Miss ratios of LFU, LRU, and Redis switching from LFU to LRU at 40 hrs for the Cloudphysics `w96` ***trace [24].***

LFU, but which reduces its metadata memory usage. Two key optimizations are:

- An object is allocated a 24-bit metadata field to store any required data pertaining to the currently configured eviction policy. For LRU, this is the last access time. For LFU, this is the last access time with reduced precision (16 bits) and the logarithmic frequency count (8 bits). There is only one such field, so its content can only pertain to one eviction policy at a time. On a policy switch, the content gets updated to that of the new policy upon access to the object. As a result, the behavior of the cache may deviate from the expected behavior for a period of time.
- Redis does not maintain an LRU or LFU stack. To select an object for eviction, a number of objects (5 by default) are randomly selected and the oldest (in the case of LRU) or least frequently accessed (in the case of LFU) is selected for eviction.[2] This method of eviction policy approximation causes the observed miss ratio to deviate from that of exact implementations of the LRU or LFU eviction policies.

To observe the effectiveness of Redis' ability to switch its eviction policy between LRU and LFU, we instantiated a Redis v8.0.1 cache of size 500MiB, initially configured with the LFU eviction policy,[3] and measured its miss ratio compared to exactly implemented LFU and LRU caches of the same size. Fig. 1 shows the results as obtained over the duration of the `w96` trace in the Cloudphysics dataset [24]. At time 40 hours, the Redis eviction policy is switched from LFU to LRU. The miss ratio of the Redis cache resembles that of LFU before 40 hours with a period of high error between hours 18 and 40, due to its approximate implementation of LFU. After 40 hours, the miss ratio of the Redis cache follows that of LRU with an error of roughly 5%.[4]

[2]This method would allow Redis to support some other eviction policies with minimal changes (e.g., in the case of FIFO, maintaining the object's entry time into the cache). However, adapting this eviction strategy to more complex policies (e.g., ARC [20], LHD [11], LIRS [29], 2Q [18], S3-FIFO [10], etc.) would likely be challenging.

[3]We use `allkeys-lfu` for LFU and `allkeys-lru` for LRU.

[4]Interestingly, repeating this experiment, we noticed different results each time, which can be attributed to Redis' eviction policy approximation (i.e., randomly selecting N objects as candidates for eviction).

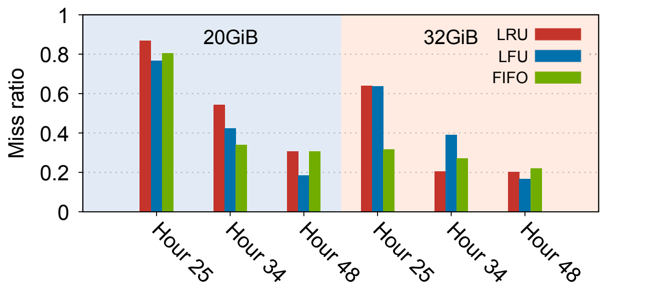

Figure 2: Hourly LRU, LFU, and FIFO miss ratios for the Cloudphysics `w24` *trace [24].*

2.2 SHARDS sampling

Waldspurger et al. describe a sampling technique called *SHARDS*, originally designed for use in efficient MRC generation [24].[5] SHARDS samples a stream of incoming cache accesses using a configurable sampling rate R. On each access, SHARDS computes $T_i = hash(K) \bmod P$, where K is the access key and P is a large static number (the authors use $P = 2^{24}$). If $T_i < RP$, the access is sampled. An important characteristic of SHARDS is that once a key is sampled, it will always be sampled.

2.3 Selecting the optimal eviction policy

The optimal[6] eviction policy for a workload can change over time. Fig. 2 shows the miss ratios for the LRU, LFU, and FIFO eviction policies for the Cloudphysics `w24` trace [24] for caches of size 20GiB and 32GiB. For a cache of size 20GiB, at 25 hours, the optimal eviction policy is LFU. At 34 hours, this changes to FIFO. Finally, at 48 hours, this changes back to LFU. Selecting an eviction policy is further complicated as the optimal eviction policy changes depending on the cache's allocated size. For the same Cloudphysics `w24` trace, using a cache size this time of 32GiB, at 25 hours, the optimal eviction policy is FIFO, at 34 hours, this changes to LRU, and finally at 48 hours, it changes to LFU.

3 PaperCache

In §2.3, we showed that the selection of the appropriate eviction policy on a per-workload basis can have significant benefits on performance. In this section, we present PaperCache, a novel in-memory cache that can dynamically switch between any eviction policy at runtime. Fig. 3 depicts PaperCache's architecture. It has three notable design elements. First, PaperCache maintains a full stack of object metadata for the currently active eviction policy (e.g., LRU in the figure). This stack is used to perform evictions (exactly) according to the active policy, and hence the stack would

[5]Although, SHARDS has two variants: *fixed-rate* and *fixed-size*, we only describe the fixed-rate variant as that is what is used in PaperCache.
[6]We define "optimal" as the eviction policy with the lowest miss ratio.

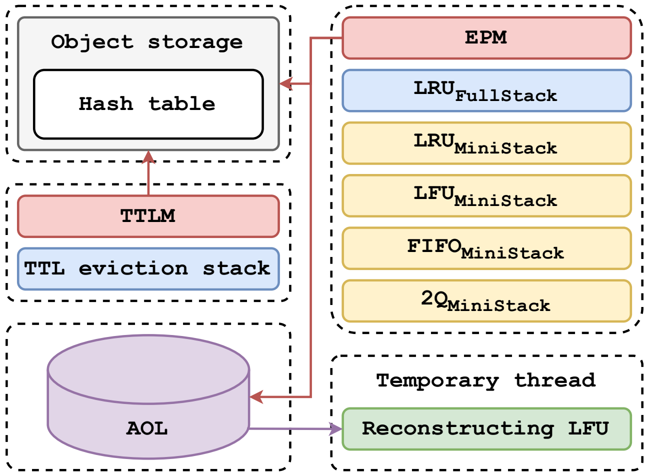

Figure 3: An overview of PaperCache (LRU). Sections shown in dashed rectangles run on separate threads.

be ordered by access recency for LRU, access frequency for LFU, and so on. The stack is updated and cache evictions are handled asynchronously by an *eviction policy manager* (**EPM**) running on a separate worker thread.

Second, for each configured eviction policy, PaperCache maintains an approximate stack of metadata, referred to as a *MiniStack*. The SHARDS fixed-rate sampling algorithm [24] is used to decide which objects are represented in the MiniStacks, thus significantly reducing their memory overheads. A MiniStack is used to temporarily decide which objects to evict, if necessary, while the cache is switching policies.

Finally, metadata of each access is streamed to disk in an *append-only-log* (**AOL**). The AOL is used to reconstruct the full stack of a target eviction policy when PaperCache switches policies.[7] This reconstruction is performed by a separate temporary thread. The AOL contains a sliding window of access metadata (default = 7 days) to limit stack reconstruction time.

When an existing object is accessed, or when a new object is inserted, the object's metadata is asynchronously sent to the EPM which then: (i) updates the full stack of the current eviction policy, (ii) updates all MiniStacks if sampled, and (iii) pushes the access metadata to the AOL. The EPM also asynchronously evicts objects from the cache (along with their metadata) when the cache's used size exceeds its configured size. Similar to prior techniques [9], PaperCache performs evictions lazily, which may cause short periods of time wherein the cache's used size exceeds its configured size (although we found this to be negligible when processing real-world access traces, which we demonstrate in §4.2).

[7]In the common case, we only store each access's key as its metadata in the AOL, however, some eviction policies may require more information about each access to perform reconstruction (e.g., the size of the access). In such cases, all required metadata of each access is saved to the AOL.

However, performing these operations asynchronously reduces access latencies as eviction policy stack operations occur off the main thread.

When the eviction policy is switched from policy P to policy P', PaperCache performs the following actions:

(1) Pauses writes to the AOL and buffers subsequent access' metadata in memory.
(2) Starts to use the MiniStack of P' for eviction decisions.
(3) Releases the full stack of P from memory.
(4) Reconstructs the full stack of P' using the AOL.
(5) Starts to use the full stack of P' for eviction decisions.
(6) Flushes buffered access' metadata to P' and the AOL, and resumes subsequent writes to the AOL.

These actions are performed asynchronously while the cache is servicing new accesses. If cache evictions must occur while the stack is being populated, objects are evicted using the MiniStack of P'. The cache therefore has a short period of approximate behavior until the full stack is reconstructed. In §4.6, we show that a cache subject to evictions from a policy's MiniStack exhibits a miss ratio within 1% of that when it is subject to the policy's full stack for a period of up to 2.7 hours, on average.

To handle TTLs, PaperCache uses a priority queue of object expiry times. This queue is managed by a separate time-to-live manager (**TTLM**) running on its own thread. The queue is checked every millisecond, and expired objects are removed from the cache as well as the metadata structures. (Expired objects are never returned to clients.)

Automatically switching eviction policies. Each PaperCache MiniStack is analogous to a single simulation in MiniSim [22]. The MiniStacks can therefore be used, at no additional cost, to periodically identify which eviction policy incurs the lowest miss ratio. This allows PaperCache to support an *auto eviction policy* where it automatically switches to the eviction policy with the lowest miss ratio. We found that performing this switch every 1 hour is effective.

4 Evaluation

In this section, we evaluate PaperCache by examining: (i) the accuracy of its policy implementations compared to ideal implementations after switching between policies (§4.1), (ii) the ability of its lazy eviction scheme to keep the memory consumed by the cached objects within the configured cache size (§4.2), (iii) its metadata memory overhead (§4.3) and latency performance (§4.4) compared to Redis, (iv) its CPU usage when switching between eviction policies (§4.5), (v) the duration for which MiniStacks provide reasonably accurate policy evictions (§4.6), and (vi) its behavior when automatically switching eviction policies (§4.7). In all experiments, PaperCache is configured with 8 eviction policies: LFU, FIFO, LRU, MRU, 2Q [18], S3-FIFO [10], CLOCK, and SIEVE [21].[8] We compare PaperCache to Redis in our evaluation as it is, to our knowledge, the only in-memory cache that supports eviction policy switching. To perform our evaluations, we used access traces from Twitter [3], Cloudphysics [24], IBM [25], Tencent CBS [26], Alibaba [27], and Wikipedia [28].[9]

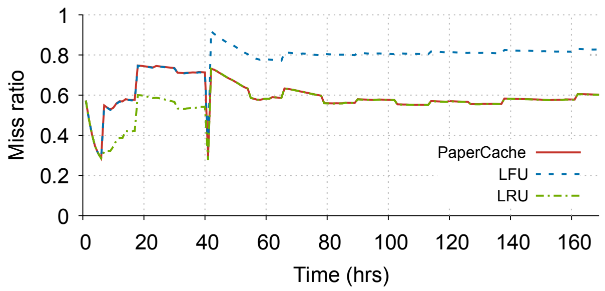

Figure 4: Miss ratios of LFU, LRU, and PaperCache switching from LFU to LRU at 40 hrs for the Cloudphysics `w96` *trace [24].*

All experiments were done on Ubuntu 24.0.2 with an Intel i9-13900KS (24 cores) with 128GB of DDR5–4200MHz DRAM. PaperCache was implemented in Rust v1.89.0 and uses jemalloc as its memory allocator.[10] Our implementation of PaperCache is open-sourced at https://papercache.io.

4.1 Policy switching accuracy

To examine the effectiveness of PaperCache's policy switching, we configured an instance of PaperCache of size 500MiB with the LFU eviction policy and show its performance compared to exact LFU and LRU caches of the same size for the `w96` trace in the Cloudphysics dataset [24], similar to the experiment we performed for Redis in §2.1. At time 40 hours, we switch PaperCache's policy from LFU to LRU and reset the hit and miss counters of the caches. Fig. 4 shows the results. We observe that before 40 hours, PaperCache closely tracks the miss ratio of LFU. After 40 hours, it switches to LRU and immediately closely tracks the miss ratio of LRU as well. (One can compare these results to those of Redis in Fig. 1.) Similar to Redis, PaperCache's eviction policy switching is a passive configuration change and we noticed no measurable effect on access throughput or latency when switching the eviction policy at runtime.

4.2 Lazy eviction performance

PaperCache's lazy eviction scheme (where evictions occur off the main thread) may result in short periods of time wherein the memory used by cached objects exceeds the configured maximum cache size (as `SET` accesses are not blocked for evictions). Here, we evaluate the resulting overhead of using

[8]We use $K_{in} = 0.25$ and $K_{out} = 0.5$ for 2Q, and $|S| = 10\%$ for S3-FIFO.
[9]We use the recommended traces in the Twitter dataset [3, 30].
[10]PaperCache uses the same version of jemalloc as Redis v8.0.1.

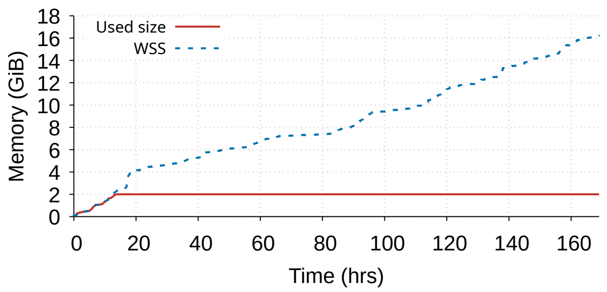

Figure 5: PaperCache used size versus WSS for the Cloudphysics* `w96` *trace [24].

Table 1: Memory overheads of Redis and PaperCache storing between* 1,000 *and* 1,000,000 *unique objects.

	1,000	10,000	100,000	1,000,000
Redis v8.0.1	15MiB	43MiB	176MiB	270MiB
PaperCache	12MiB	53MiB	195MiB	459MiB

lazy evictions. We initialized a PaperCache instance of size 2GiB and the LRU eviction policy, and we then applied the accesses in the Cloudphysics `w96` access trace [24]. We measure the aggregate size of objects in the cache versus the access trace's working set size (**WSS**) over the duration of the experiment, where the WSS is the aggregate size of all unique objects accessed thus far. Fig. 5 shows the results. Measured after the WSS exceeds 2GiB, PaperCache's used size is, on average, within 0.01% that of the configured maximum size, up to a maximum of 0.15%.

4.3 Memory overhead

PaperCache's support of multiple eviction policies and the switching between them comes at the cost of higher memory overhead to maintain the full metadata stack (for the eviction policy in place) and N MiniStacks, where N is the number of configured eviction policies. To measure the extent of this overhead, we compare PaperCache with Redis v8.0.1 [16], both configured to use 2GiB and LRU. We inserted between 1,000 and 1,000,000 unique objects, using object sizes such that the cached objects consume 1GiB in aggregate (so no evictions would occur), and then measured the high water mark (**HWM**) of the cache's residency set size. Tbl. 1 shows the overheads as the HWM minus 1GiB. PaperCache has between 10.8% and 70% higher memory overhead than Redis when storing between 100,000 and 1,000,000 unique objects, respectively, with 8 configured MiniStacks and a MiniStack sampling rate of 0.1%. We note that PaperCache uses slightly less memory than Redis when storing 1,000 objects which is attributed to smaller initial allocations for its data structures.

4.4 Latency performance

We experimentally compare the performance of PaperCache and Redis by measuring the latencies of `GET` and `SET` accesses. For this, we instantiated PaperCache and Redis caches of size 2GiB under LRU and performed the accesses in all access traces in the Cloudphysics dataset [24] (106 access traces comprising of over 2 billion total accesses) using 4 concurrent clients for each cache. Tbl. 2 shows the p99, p99.9, and p99.99 percentile access latencies. We found that PaperCache has 30.5% and 43.4% lower p99.99 latencies for `GET` and `SET` accesses, respectively. These lower latencies are due to evictions in PaperCache occurring off the main thread.

Table 2: Access latency percentiles of Redis and PaperCache for all traces in the Cloudphysics dataset [24].

	p99 (μs)		**p99.9 (μs)**		**p99.99 (μs)**	
	GET	**SET**	**GET**	**SET**	**GET**	**SET**
Redis v8.0.1	116	595	298	1,327	600	1,889
PaperCache	66	523	177	806	417	1,070

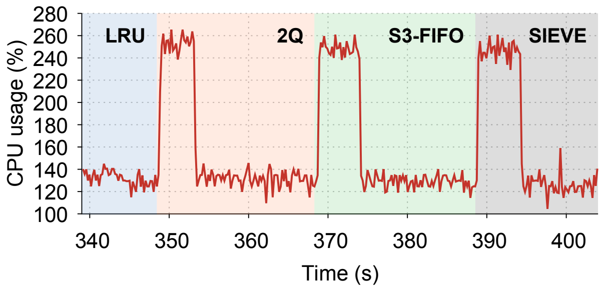

Figure 6: Total CPU usage of PaperCache (initially under LRU) when switching to the 2Q, S3-FIFO, and SIEVE eviction policies for the Cloudphysics* `w07` *trace [24].

4.5 CPU usage

We examine PaperCache's increased CPU usage while reconstructing the stack of a new policy by performing policy switches when processing the `w07` trace in the Cloudphysics dataset [24]. Fig. 6 shows the total CPU usage of PaperCache (allocated 64GiB) as it switches from LRU to 2Q, S3-FIFO, then SIEVE. Although PaperCache has increased CPU usage when a policy stack is being reconstructed after a policy switch, we note that the extra CPU overhead occurs off the main thread, so there is no impact on access latency.

4.6 MiniStack efficacy duration

During an eviction policy switch, while a MiniStack is being used to perform evictions, the miss ratio of the cache can deviate from that of the exact eviction policy as evicted objects are removed from the MiniStack and only sampled accessed objects are added. To evaluate the efficacy of a MiniStack in approximating the evictions of its corresponding policy, we initialized two LRU caches of equal size: one subject to evictions from a full LRU stack and the other from an LRU MiniStack. We apply the same access trace to each cache and measure the duration between when the first eviction occurs to when the miss ratios of the two caches differs by $\geq 1\%$,

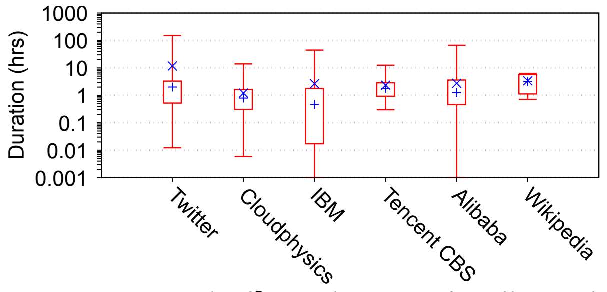

Figure 7: MiniStack efficacy durations for all considered datasets. The bottom and top lines identify the minimum and maximum results, respectively. The bottom and top of each box are the 25^{th} and 75^{th} percentile results, respectively. The $\times$ and + symbols indicate the mean and median results, respectively.

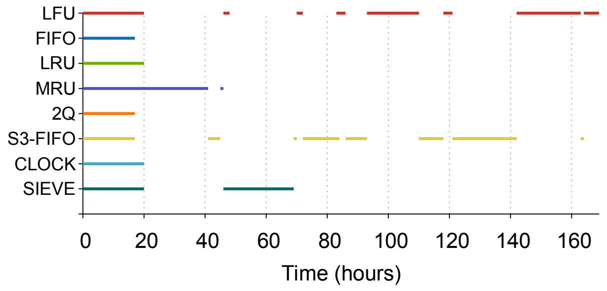

Figure 8: Instances where each configured PaperCache eviction policy achieves the lowest miss ratio for the `w42` trace in the Cloudphysics dataset [24].

which we refer to as the MiniStack's *efficacy duration*. Each cache is allocated 10% of the access trace's WSS.

Fig. 7 demonstrates the results of this experiment for 873 workloads that include over 189 billion accesses. Here, we found the average and median efficacy durations to be 2.7 and 1.1 hours, respectively. The largest access trace is Twitter's `cluster18` trace [3] which includes 12.6 billion accesses. Experimentally, we found that PaperCache reconstructs a full stack at a rate of 18.2 million accesses per second, on average, and thus PaperCache reconstructs the full stack for `cluster18` in 11.5 minutes. As this is significantly less than the MiniStack efficacy duration, PaperCache's miss ratio will accurately track that of exact implementations of its eviction policies during a policy switch.

4.7 Automatic policy switching behavior

We evaluate PaperCache's ability to automatically switch eviction policies. For this, we configure PaperCache to switch to the corresponding policy of the MiniStack with the lowest miss ratio every 1 hour. We used the Cloudphysics `w42` access trace [24] which spans 168 hours, and we configured PaperCache with a size of 10% of the trace's WSS. Throughout this access trace, PaperCache (initially under LRU) performs 17 policy switches, reducing the miss ratio compared to LRU by up to 9.7%. Tbl. 3 shows the percentages of the total trace duration where each eviction policy achieves the lowest and strictly lowest miss ratio (where achieving the "strictly lowest" miss ratio indicates it achieves a miss ratio lower than that of any other eviction policy). The LFU eviction policy has the best performance for the largest duration of the access trace, achieving the lowest and strictly lowest miss ratio for 43.2% and 29.6% of the total trace duration, respectively. Fig. 8 shows the instances where each policy achieves the lowest miss ratio over the duration of the trace.

PaperCache's automatic eviction policy switching can considerably reduce the miss ratio of the cache. Fig. 9 shows the miss ratio savings achieved by PaperCache when performing automatic eviction policy switching compared to statically configured eviction policies for the `w02` access trace in the Cloudphysics dataset [24]. We configured PaperCache with an allocated size of 10% of the access trace's WSS. PaperCache reduced its miss ratio by between 8.2% and 48.5%.

Table 3: % time each policy achieves the lowest and strictly lowest miss ratios, where "strictly lowest" means it is not tied, for the Cloudphysics `w42` access trace [24].

Policy	% time min. MR	% time strictly min. MR
LFU	43.2	29.6
FIFO	10.1	0
LRU	11.8	0
MRU	24.9	13
2Q	10.1	0
S3-FIFO	42	31.4
CLOCK	11.9	0
SIEVE	25.4	12.4

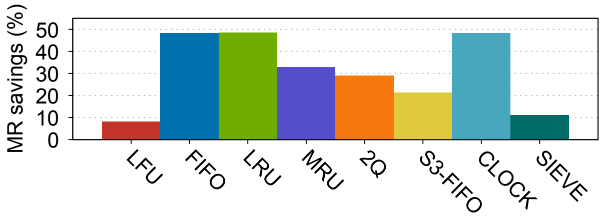

Figure 9: Miss ratio savings by performing eviction policy switching compared to statically assigned eviction policies for the Cloudphysics `w02` trace [24].

5 Concluding remarks

In this paper, we introduced PaperCache, a novel in-memory cache design which supports the dynamic switching between any eviction policy at runtime. We demonstrated that a workload's optimal eviction policy can change over time, reinforcing the need for PaperCache. We introduced our novel eviction policy switching technique through the use of MiniStacks and show how they can be leveraged to perform automatic eviction policy switching, which we showed can reduce PaperCache's miss ratio by up to 48.5%. In the future, we plan to reduce PaperCache's memory overhead and extend it to support admission policies.

References

[1] Jhonny Mertz and Ingrid Nunes. Understanding Application-Level Caching in Web Applications: A Comprehensive Introduction and Survey of State-of-the-Art Approaches. *ACM Computing Surveys*, 50 (6):1–34, November 2017. ISSN 0360-0300.

[2] Ao Wang, Jingyuan Zhang, Xiaolong Ma, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Vasily Tarasov, Feng Yan, and Yue Cheng. InfiniCache: Exploiting ephemeral serverless functions to build a cost-effective memory cache. In *Proc. Conf. on File and Storage Technologies (FAST'20)*, pages 267–281, February 2020. ISBN 978-1-939133-12-0.

[3] Juncheng Yang, Yao Yue, and K. V. Rashmi. A Large-Scale Analysis of Hundreds of In-Memory Key-Value Cache Clusters at Twitter. *ACM Transactions on Storage*, 17(3):1–35, August 2021.

[4] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling Memcache at Facebook. In *Proc. Symp. on Networked Systems Design and Implementation (NSDI'13)*, pages 385–398, April 2013. ISBN 978-1-931971-00-3.

[5] Juncheng Yang, Yao Yue, and Rashmi Vinayak. Segcache: A memory-efficient and scalable in-memory key-value cache for small objects. In *Proc. Symp. on Networked Systems Design and Implementation (NSDI'21)*, pages 503–518, April 2021. ISBN 978-1-939133-21-2.

[6] Yue Cheng, Aayush Gupta, and Ali R. Butt. An in-memory object caching framework with adaptive load balancing. In *Proc. of the European Conf. on Computer Systems (EuroSys'15)*, pages 1–16, April 2015. ISBN 9781450332385.

[7] Jaewon Kwak, Eunji Hwang, Tae-Kyung Yoo, Beomseok Nam, and Young-Ri Choi. In-memory caching orchestration for Hadoop. In *Proc. Intl. Symp. on Cluster, Cloud and Grid Computing (CCGrid'16)*, pages 94–97, May 2016.

[8] Kia Shakiba, Sari Sultan, and Michael Stumm. Kosmo: Efficient online miss ratio curve generation for eviction policy evaluation. In *Proc. Conf. on File and Storage Technologies (FAST'24)*, pages 89–105, February 2024. ISBN 978-1-939133-38-0.

[9] Hojin Park, Ziyue Qiu, Gregory R. Ganger, and George Amvrosiadis. Reducing cross-cloud/region costs with the auto-configuring MACARON cache. In *Proc. Symp. on Operating Systems Principles (SOSP'24)*, pages 347–368, November 2024.

[10] Juncheng Yang, Yazhuo Zhang, Ziyue Qiu, Yao Yue, and Rashmi Vinayak. FIFO queues are all you need for cache eviction. In *Proc. Symp. on Operating Systems Principles (SOSP'23)*, pages 130–149, October 2023. ISBN 9798400702297. doi: 10.1145/3600006.3613147.

[11] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. LHD: Improving cache hit rate by maximizing hit density. In *Proc. Symp. on Networked Systems Design and Implementation (NSDI'18)*, pages 389–403, April 2018. ISBN 978-1-939133-01-4.

[12] Aaron Blankstein, Siddhartha Sen, and Michael J. Freedman. Hyperbolic caching: Flexible caching for web applications. In *Proc. USENIX Annual Technical Conf. (USENIX ATC'17)*, pages 499–511, July 2017. ISBN 978-1-931971-38-6.

[13] Bin Fan, David G. Andersen, and Michael Kaminsky. MemC3: Compact and concurrent MemCache with dumber caching and smarter hashing. In *Proc. Symp. on Networked Systems Design and Implementation (NSDI'13)*, pages 371–384, April 2013. ISBN 978-1-931971-00-3.

[14] Ankita Kejriwal, Arjun Gopalan, Ashish Gupta, Zhihao Jia, Stephen Yang, and John Ousterhout. SLIK: Scalable low-latency indexes for a key-value store. In *Proc. USENIX Annual Technical Conf. (USENIX ATC'16)*, pages 57–70, June 2016. ISBN 978-1-931971-30-0.

[15] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. KV-Direct: High-performance in-memory key-value store with programmable NIC. In *Proc. Symp. on Operating Systems Principles (SOSP'17)*, pages 137–152, October 2017. ISBN 9781450350853. doi: 10.1145/3132747.3132756.

[16] Redis Labs. Redis. https://redis.io.

[17] Memcached. Memcached. https://memcached.org.

[18] Theodore Johnson, Dennis Shasha, et al. 2Q: A low overhead high performance buffer management replacement algorithm. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB'94)*, pages 439–450, September 1994.

[19] S. Min, D. Lee, C. Kim, J. Choi, J. Kim, Y. Cho, and S. Noh. LRFU: A Spectrum of Policies that Subsumes the Least Recently Used and Least Frequently Used Policies. *IEEE Transactions on Computers*, 50(12):1352–1361, December 2001. ISSN 1557-9956. doi: 10.1109/TC.2001.970573.

[20] Nimrod Megiddo and Dharmendra S. Modha. ARC: A self-tuning, low overhead replacement cache. In *Proc. Conf. on File and Storage Technologies (FAST'03)*, pages 115–130, March 2003.

[21] Yazhuo Zhang, Juncheng Yang, Yao Yue, Ymir Vigfusson, and K.V. Rashmi. SIEVE is simpler than LRU: an efficient Turn-Key eviction algorithm for web caches. In *Proc. Symp. on Networked Systems Design and Implementation (NSDI'24)*, pages 1229–1246, April 2024.

[22] Carl Waldspurger, Trausti Saemundsson, Irfan Ahmad, and Nohhyun Park. Cache modeling and optimization using Miniature Simulations. In *Proc. USENIX Annual Technical Conf. (USENIX ATC'17)*, pages 487–498, July 2017. ISBN 978-1-931971-38-6.

[23] 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 *Proc. Symp. on Operating Systems Design and Implementation (OSDI'20)*, pages 753–768, November 2020.

[24] Carl A. Waldspurger, Nohhyun Park, Alexander Garthwaite, and Irfan Ahmad. Efficient MRC construction with SHARDS. In *Proc. Conf. on File and Storage Technologies (FAST'15)*, pages 95–110, February 2015. ISBN 978-1-931971-201.

[25] Ohad Eytan, Danny Harnik, Effi Ofer, Roy Friedman, and Ronen Kat. It's time to revisit LRU vs. FIFO. In *Proc. Workshop on Hot Topics in Storage and File Systems (HotStorage'20)*, July 2020.

[26] Tencent. Tencent CBS. https://intl.cloud.tencent.com/product/cbs.

[27] Alibaba. Alibaba Block Traces. https://github.com/alibaba/block-traces.

[28] Wikimedia Foundation. Wikipedia CDN Traces. https://wikitech.wikimedia.org/wiki/Analytics/Data_Lake/Traffic/Caching.

[29] Song Jiang and Xiaodong Zhang. LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Proc. Intl. Conf. on Measurement and Modeling of Computer Systems (SIGMETRICS'02)*, pages 31–42, June 2002. ISBN 1581135319.

[30] Twitter. Anonymized Cache Request Traces from Twitter Production. https://github.com/twitter/cache-trace.