SL-Cache: Selective Learning Cache Eviction with Priority Retention for Hot Objects

Yu-Ang Zhang[1], Yigui Yuan[1], Peiquan Jin[1(✉)], Hongtao Wang[1], Xiaoliang Wang[2], and Shouhong Wan[1]

[1] University of Science and Technology of China, Hefei, China
jpq@ustc.edu.cn
[2] Hefei University of Technology, Hefei, China

Abstract. Learning-based cache eviction algorithms, particularly object-level approaches, have demonstrated superior reduction in miss ratio by making fine-grained eviction decisions. However, this accuracy often comes at a high computational cost, as existing methods typically apply complex prediction models uniformly to all objects and struggle to quickly differentiate hot from cold items. In this paper, we identify two critical insights: (1) not all objects necessitate precise prediction, especially one-hit wonders lacking rich historical features, and (2) effective sampling must rapidly protect hot objects and identify cold eviction candidates. Based on these insights, we propose **SL-Cache** (Selective Learning Cache), a novel object-level learning eviction algorithm that optimally balances miss ratio and throughput. SL-Cache introduces two key innovations: first, it bypasses costly model predictions for low-utility one-hit wonders by employing precomputed utility estimates, substantially reducing computational overhead. Second, it implements an adaptive, hotness-aware sampling strategy that dynamically demotes objects based on refined access patterns, ensuring hot objects are retained while non-hot candidates are efficiently selected for prediction or eviction. Our extensive experimental evaluation across ten state-of-the-art eviction algorithms, using diverse production and benchmark traces, demonstrates that SL-Cache consistently achieves a superior balance. Specifically, SL-Cache reduces the miss ratio by 11.90% relative to LRU, which is 1.1x higher than the state-of-the-art 3L-Cache, while simultaneously delivering 2x the throughput of 3L-Cache. Furthermore, SL-Cache achieves the highest throughput among all object-level learning algorithms.

Keywords: Cache eviction · Object-Level Learning · Machine Learning

1 Introduction

Cache is a fundamental component in key-value stores [19] and database management systems (DBMSs) [11,14], bridging the significant performance gap

H. Jung et al. (Eds.): DASFAA 2026, LNCS 16535, pp. 589–605, 2026.
https://doi.org/10.1007/978-981-92-0363-5_36

between main memory and storage devices. Effective caching relies on high hit rates, as cache hits provide low-latency data access, while misses incur costly storage I/O. Consequently, minimizing the cache miss rate is a critical objective. As the heart of the cache system, the eviction algorithm dictates which objects are evicted and thus directly governs overall efficiency. Traditional eviction algorithms typically employ heuristics based on recency [7–9,13], frequency [6], or their hybrid combinations [10,12]. Recent advancements have introduced techniques such as lazy promotion and quick demotion [21,24] to enhance the adaptability of heuristic-based approaches to observed workload patterns. However, this reliance on fixed feature combinations constitutes a fundamental limitation, as it prevents traditional algorithms from maintaining high efficiency across diverse workloads.

Recently, learning-based eviction algorithms have emerged as a promising alternative since they adapt to varying workload characteristics [22,23]. These algorithms can be categorized into four primary paradigms:

(1) *Distribution learning* algorithms [2] model object request probability using historical heuristic features to inform eviction decisions.
(2) *Pattern learning* algorithms [15,18,23] employ multiple heuristic algorithms as experts, dynamically switching between them based on workload patterns.
(3) *Group-level learning* algorithms [20] cluster objects into groups using heuristic features(e.g., write time) and predict utility at the group level, reducing computational overhead.
(4) *Object-level learning* algorithms [4,17,25] represent the most fine-grained approach, predicting next access times for individual objects to approximate Belady's OPT algorithm [3].

Among all these paradigms, object-level learning algorithms consistently achieve superior reductions in miss ratio across diverse real-world workloads, owing to their comprehensive feature utilization and granular prediction capabilities.

Among these paradigms, object-level learning algorithms have demonstrated the potential to achieve the lowest miss ratios by leveraging multiple features and operating at the finest granularity. However, this accuracy comes at the cost of significant computational overhead. For object-level learning algorithms, the sampling strategy significantly affects the eviction algorithm's performance because it determines which candidates are selected for eviction. Previous works [17,25] keep a relatively large "*prediction ratio*" (Average prediction times per eviction) to guarantee the quality of eviction decision. A higher prediction ratio yields a higher-quality eviction but incurs a higher computational cost. To tackle this challenge, we identify two critical insights from an object-centric perspective:

(1) Not all objects need precise prediction. A significant fraction of objects, particularly "*one-hit wonders*" [21] (i.e., objects accessed only once after insertion), lack the rich historical access features (e.g., reuse distance, frequency) required by complex learning models. Forcing these objects through the pre-

diction pipeline yields little benefit, as their eviction utility is low and can be estimated using simpler methods.

(2) Efficient sampling must rapidly differentiate hot from cold objects. The sampling strategy's goal is twofold: protect hot objects from being sampled (and thus from potential eviction) and quickly identify cold objects as eviction candidates. Traditional heuristics are often too slow to react to dynamic patterns, leading to either the premature eviction of hot objects or unnecessary predictions on them.

Based on these insights, we propose **S**elective **L**earning **Cache** (**SL-Cache**), a novel object-level learning algorithm that strategically *bypasses* costly predictions for low-utility and hot objects. Rather than solely optimizing the prediction model, our core contribution lies in selectively applying predictions by efficiently identifying which objects do not warrant the prediction cost. More specifically, SL-Cache introduces two key innovations: First, to handle one-hit wonders efficiently, SL-Cache directly uses precomputed estimates of their low utility in eviction decisions. This eliminates the need to invoke the primary prediction model for these objects, thereby substantially reducing computational overhead. Second, an adaptive, hotness-aware sampling strategy that dynamically demotes objects based on access patterns, ensuring that hot objects are retained while non-hot candidates are efficiently selected for prediction or eviction. Together, these designs allow SL-Cache to significantly reduce prediction overhead while maintaining a low miss ratio.

Briefly, we make the following contributions in this paper:

- We conduct a critical analysis of object-level learning algorithms from an object-centric perspective, highlighting scenarios in which learning models increase computational overhead without corresponding reductions in miss ratios. We propose the first approach that bypasses model predictions for specific objects and scenarios, thereby improving throughput while maintaining a low miss ratio.
- We design and integrate an adaptive, hotness-aware sampling mechanism that dynamically adjusts object promotion and demotion during fine-tuning, effectively insulating hot objects from eviction and efficiently identifying cold candidates.
- We conducted an extensive experimental evaluation of SL-Cache against 10 state-of-the-art eviction algorithms using diverse production and benchmark traces. The results demonstrate that SL-Cache consistently achieves a superior trade-off between miss ratio and throughput. In particular, SL-Cache reduces the miss ratio by 11.90% compared with LRU. This reduction is 1.1× greater than that achieved by the state-of-the-art 3L-Cache, while SL-Cache delivers 2× the throughput of 3L-Cache.

The remainder of the paper is structured as follows. Section 2 discusses related work and research motivations. Section 3 presents the detailed designs of SL-Cache. Section 4 reports the experimental results, and finally Sect. 5 concludes the paper and discusses future work.

2 Related Work and Motivations

2.1 Object-Level Learning for Cache Eviction

To achieve a lower miss ratio and enhance adaptability to diverse workloads, an emerging research paradigm integrates machine learning (ML) techniques into cache eviction algorithm design [16,17,25]. Among these learning algorithms, *object-level learning* has demonstrated superior performance across a broad range of real-world workloads by making fine-grained eviction decisions for individual objects.

Object-level learning algorithms typically predict either the next access time of objects [4,17,25] or custom metrics designed to quantify object utility. The foundational design philosophy of these algorithms aims to emulate the prescient decision-making process of Belady's OPT algorithm. Whereas OPT leverages perfect future knowledge to evict the object with the longest future reuse distance, object-level learning algorithms employ ML models trained on historical access patterns. These models use heuristic features, such as reuse distance (e.g., the number of intervening requests between accesses to the same object), age, and frequency, to predict future reuse intervals or object utility. However, the computational cost of model training and prediction results in higher latency and degraded overall throughput. To manage this trade-off between prediction quality and computational cost, existing object-level learning algorithms employ a **sampling strategy** to select a subset of objects for prediction. The intensity of this process can be measured by the **prediction ratio**, defined as the average number of predictions performed per eviction. For instance, LRB [16] randomly samples 64 objects from the cache for prediction during one eviction. HALP [17] samples four objects from the tail of LRU and uses three rank-based predictions for one eviction. 3L-Cache [25] deploys a bio-sampling strategy based on LRU and frequency of evicted objects to select a batch of candidates for prediction, subsequently evicting half the batch size. Current object-level learning approaches face two primary limitations stemming from their prediction and sampling strategy: (1) These algorithms generate predictions for every object in the same way, regardless of analyzing the necessity of precise predictions for every object. (2) These algorithms may not sample the cold objects quickly enough, and cause cold objects to accumulate in the cache since they directly deploy a heuristic algorithm or features.

2.2 Motivations

The Case of One-Hit Wonders. The prevalence of objects accessed only once after admission within cache workloads, often termed **one-hit wonders** [21]. Analysis further reveals that these objects constitute a significant fraction of the items evicted by the optimal Belady's algorithm, often being discarded relatively early in their cache lifetime, as shown in Fig. 1. While heuristic algorithms implicitly handle these objects (evicting them at the tail in LRU), their impact on the efficiency of *object-level learning* algorithms warrants specific attention. Object-level learning algorithms predict an object's utility from historical request

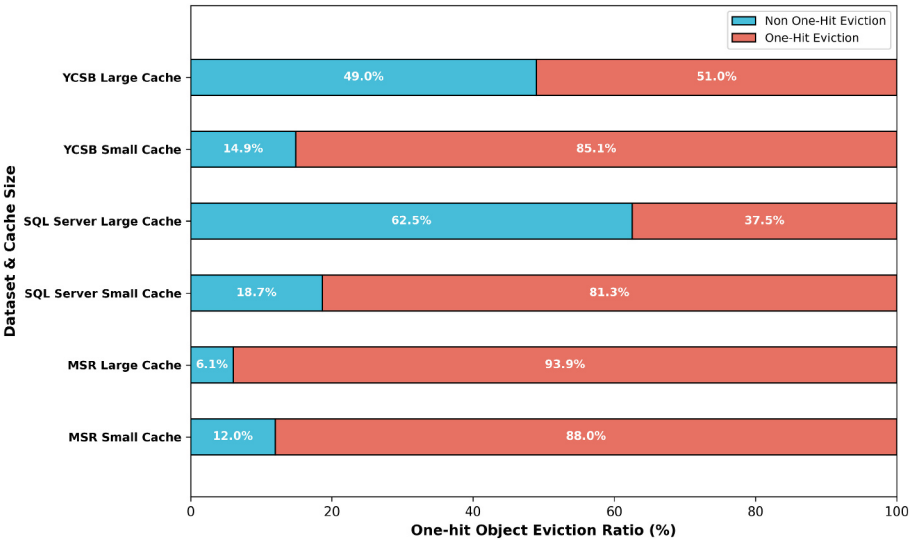

Fig. 1. One-hit object prediction ratio in Belady's OPT algorithm.

features. However, one-hit wonders offer only age as a dynamic feature, lacking a history for models to rely on. Hence, predicting these objects with complex models is inefficient. This motivates estimating their eviction priority using simpler, potentially precomputed methods, enabling selective bypass of the main prediction model. Such bypassing significantly cuts computational overhead for these numerous objects while preserving cache performance.

Hot-Oriented Sampling Strategy. Traditional heuristic-based Sampling Strategies often fail to adequately protect hot objects (e.g., LRU is not scan-resistant and cannot retain hot objects), thereby hindering miss-ratio minimization. We propose a fundamentally *hotness-oriented* Sampling Strategy which prioritizes identifying and preserving the hot working set. This requires accurately assessing an object's "hotness" using multiple features, rather than relying on single-dimensional heuristics. Integrating dynamic techniques, such as quick demotion, further enhances this by rapidly identifying cooling objects as non-hot eviction candidates. A hotness-oriented approach thus aims to both safeguard the working set for low miss ratios and efficiently provide suitable candidates for prediction or eviction.

3 Design of SL-Cache

Based on our observations and motivation, this section presents a new learning eviction algorithm, **S**elective-**L**earning **Cache** (which we term "**SL-Cache**"). We begin by outlining the structure of SL-Cache and then detail all of its components.

3.1 SL-Cache Framework

Figure 2 illustrates the structure of SL-Cache. SL-Cache processes incoming requests by first performing a lookup in the Hotness-Oriented Three-Layer Structure(§3.2) to determine whether the request hit. Cache hits and metadata movement during evictions trigger the data collection process, gathering essential information for workload adaptation. Collected access information fuels Learning Framework (§3.4) for periodic model retraining, pre-computation for one-hit wonders, and parameter adjustments. When the cache is full, the selective sampling strategy (§3.3) interacts with the cache structure to select eviction candidates and triggers comparison-based eviction (§3.5) to evict objects that are likely to be accessed in the future.

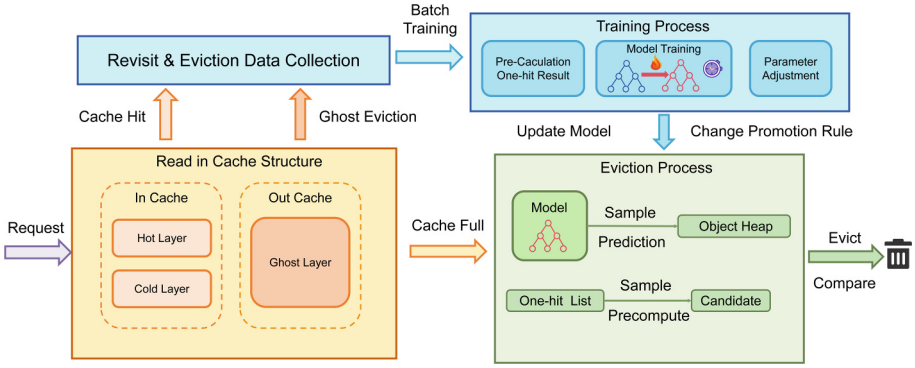

Fig. 2. Overview of SL-Cache.

3.2 Hot-Oriented Cache Structure

The cache structure governs object management and candidate selection, thereby directly affecting performance. SL-Cache employs a hotness-oriented, three-layer structure:

- **Hot Layer** Contains frequently and recently accessed data.
  - *An list*: A FIFO list holding newly arrived objects, allocated a small and dynamic portion of the cache.
  - *Am list*: A FIFO list storing objects promoted due to high frequency and recency. Its size adapts dynamically.
- **Cold Layer**: Contains less frequently accessed multi-hit objects and one-hit wonders.
  - *Candidate list*: A LRU list holding multi-hit objects(requested more than once after insertion) demoted from the Hot Layer that are not considered hot.

  - *A1 list*: A FIFO list holding all one-hit wonders demoted from the *An list*.
- **Ghost Layer**: Only stores the metadata of evicted objects, including keys and features.
  - *Ghost list*: A FIFO list storing metadata of evicted objects, serving as the source for training data.

Object movement follows these rules upon request processing:

(1) Request Handling: On a cache hit, metadata of the requested object is updated. Promotion might occur if hit in the Cold Layer (see rule 2). On a cache miss, the new object is directly inserted into the head of the *An list*.

(2) Promotion on Hit: If a hit occurs in the cold layer, the object is promoted. If its frequency exceeds $freq\ boundary$, it moves to the head of the *Am list*; otherwise, to the head of the *Candidate list*. Hits in hot layer do not change the object's position within its list.

(3) Insertion/Eviction on Miss: A miss inserts the new object into *An list*. If the cache is full, the eviction process (§3.5) is triggered before insertion.

(4) Demotion from *An list*: If the *An list* exceeds the capacity of An, the tail object is demoted:

- To *Am list* head if its frequency $> freq\ boundary$.
- To *A1 list* head if it is a one-hit wonder.
- To *Candidate list* head otherwise.

The parameter $freq\ boundary$ is dynamically set based on the P99 frequency of recently evicted objects, serving as an adaptive threshold that reflects current workload characteristics to quickly demote potentially non-hot (i.e., not classified as hot) new objects.

3.3 Adaptive Adjustment and Sampling Strategy

The effectiveness of object-level learning depends not only on prediction accuracy but critically on the sampling strategy used to select candidates for prediction. An inadequate strategy that fails to protect hot objects or identify cold ones promptly can undermine the entire learning process.

SL-Cache integrates object adjustment (promotion/demotion) with the sampling strategy adaptively. The sampling aims to identify non-hot candidates for prediction while shielding the hot working set. Feature statistics from evicted objects inform the adjustment strategy, specifically through periodic recalculations of $freq\ boundary$ based on eviction statistics, thereby refining the definition of hotness over time.

Sampling distinguishes between new and old objects:

- **New Objects (from An tail demotion):** Objects whose frequency exceeds the dynamically updated $freq\ boundary$ are considered hot and moved to *Am list*. Others are considered non-hot candidates, sampled for prediction (if learning is active), and moved to the *A1 List* or the *Candidate list*.

– **Old Objects (from Cold Layer and Am tail):** The sampling strategy considers all objects in the *Candidate list* as potential candidates (as they are already deemed non-hot by frequency). From the *Am List*, only a small fraction (e.g., the tail 1%, empirically determined) representing objects with extremely low recency, along with objects having the lowest hit count (indicating lower combined hotness, see below), are sampled. This focuses prediction effort while protecting the core hot set in *Am List*.

The maximum number of objects sampled in a batch (from both new and old sources) is capped (default: 1% of cache size) to control overhead.

Hotness Metrics. To dynamically quantify and manage object hotness within the *Am list*, we introduce two coupled metrics: hit count and demote count. (1) The hit count is incremented by 1 upon each access, which reflects the frequency of recent access. (2) The demote count is incremented by 1 during periodic demotion scans, which represents the age of the object. An access will reset the demote count to 0. The demotion decay scan periodically adjusts the hit count by dividing it by (demote count + 1) and then adding demote count + 1. This processes is triggered when eviction process(§3.5) needs new candidates.

This dual mechanism combines frequency tracking (hit-count increments) with recency-based decay (decay is driven by the demote count, which increases over time without hits). Consequently, frequently and recently accessed objects retain a high hit count, thus adaptively reflecting their current hotness. Objects with the lowest residual hit count are prioritized for sampling from the *Am list*.

Rationale for Dynamic Demotion. The SL-Cache design premises its demotion logic on two distinct, orthogonal triggers derived from state-of-the-art frequency-based algorithms (TinyLFU [6], S3FIFO [21]): (1) a frequency saturation event (as in TinyLFU's counter halving), and (2) an eviction-driven necessity (as in S3FIFO's count reduction). We integrate both conditions into our hot-oriented design to capture complementary workload dynamics. Furthermore, the combined use of hit count and demote count introduces a controlled recency decay, ensuring that an object's hotness rapidly diminishes (e.g., an object with maximum *hit count* = 64 decays through $32 \rightarrow 8 \rightarrow 1 \rightarrow 0$ over a maximum of four non-re-accessed demotion steps), thereby accelerating the demotion of stale objects. Critically, reaccessing an object resets its *demote count* and potentially increases its *hit count* (upon a hit), providing stronger hotness reaffirmation than observed in algorithms like S3FIFO. This integrated approach combines frequency and recency factors to robustly safeguard the hot-working set and expedite the degradation of non-hot objects.

3.4 Learning Architecture

The learning architecture is responsible for model training, prediction (for multi-hit objects), and managing training data.

Model and Features. SL-Cache employs Gradient-Boosting Decision Trees (GBDT) for prediction, chosen for its inherent robustness to feature sparsity

and its independence of feature normalization. The model is trained to predict the duration until the next object access.

- *Label* The dependent variable is the logarithm of the inter-arrival time: log(next_access_time − predict_time).
- *Loss Function* Training is optimized using the Mean Squared Error (MSE) as the L_2 loss function.
- *Features* The model utilizes three categories of features to capture object usage patterns: Frequency, Age, and Reuse distances.

The GBDT framework naturally accommodates missing initial reuse distance values.

Training Data Collection. Efficiently collecting representative training data is crucial. SL-Cache employs sampling and dynamic boundary adjustment to balance training data quality with storage memory overhead.
Dynamic Boundary Setting. To limit memory usage while capturing relevant access patterns, training data is collected only for object re-accesses within a dynamically adjusted boundary, which is tracked in the *Ghost list*. The large *Ghost list* captures a longer pattern of reuse. The total number of object metadata entries maintained in the *Ghost list* is set to *object boundary* times the cache size (in objects). *object boundary* (default 2, limit to 6 to constrain the memory usage) is adjusted based on the utility of the most recently added segment (the last "cache size" entries) of the *Ghost list*. If this segment yields a sufficiently high number of useful training samples (measured by hits within it relative to total ghost hits or cache hits), the *object boundary* is expanded by 1 until it reaches the limit.
Data Sampling. For each request landing within the current boundary, SL-Cache randomly samples one object's metadata to potentially track for training. If the sampled object is accessed again within the boundary, the measured next-access distance constitutes a training sample. If it gets evicted beyond the boundary, its distance is recorded as a large capped value (boundary size + age). Training is triggered periodically when a batch of new samples (*batch size*, default setting is 65536 based on experiment result) is collected.

3.5 Eviction Process

SL-Cache employs two eviction modes: a heuristic mode used before training and a learning-based mode. The selective nature primarily refers to bypassing predictions for one-hit wonders.
Heuristic Eviction. When ML predictions are not used, eviction proceeds heuristically based solely on the cache's hot layer. When the cache is full, if *An list* exceeds the default limit, the tail of *An list* is removed, it will be directly evicted if it is a one-hit wonder, otherwise it enters the*Am list* and triggers the demotion process to find an object with the lowest hit count to evict.

Learning-Based Eviction. When the ML model is active, the sampling strategy (§3.3) identifies eviction candidates. For multi-hit candidates, we feed their features into the GBDT model to predict the next access time. These candidates are inserted into a max-heap prioritized by their predicted time. The heap is periodically refreshed. For one-hit wonder candidates, the object at the tail of the *A1 list* is selected. The predicted next access time is retrieved from pre-computed results based on its current age. The predicted time of the candidate at the top of the max-heap is compared with the predicted time of the oldest one-hit wonder. The object with the longer predicted time until next access is evicted.

This process implicitly controls the residency of one-hit wonders based on their predicted utility relative to sampled multi-hit objects, rather than using a fixed partition. The prediction ratio for SL-Cache is dynamic: it is 2 for heap evictions but drops to 0 for one-hit wonder evictions (since they use pre-computed results), resulting in an overall ratio <2, which is the static boundary set by the state-of-the-art algorithm, 3L-Cache.

4 Performance Evaluation

In this section, we evaluate SL-Cache to address two key questions:

- How does SL-Cache compare to state-of-the-art eviction algorithms in terms of miss ratio and throughput?
- How does SL-Cache achieve a lower miss ratio and higher throughput?

4.1 Experiment Setup

Traces and Cache Size. We use a wide variety of traces representing a diverse set of workloads from three dataset sources (see Table 1). We select traces with more than 5 million requests and a compulsory miss ratio below 50% to comprehensively evaluate the performance of the eviction algorithm. We select 1% of the object count (5% for SQL Server to avoid an extremely high miss ratio) as the small cache size and 10% of the object count as the large cache size on each trace to evaluate the performance of the eviction algorithm across different cache sizes.

Baselines. In this paper, we compare SL-Cache with 10 state-of-the-art cache eviction algorithms:

- *Heuristic Algorithms*: Hill-Cache [10], S3-FIFO [21], SIEVE [24], and TinyLFU [6].
- *Distribution Learning Algorithm*: LHD [2].
- *Group Learning Algorithm*: GL-Cache [20].
- *Pattern Learning Algorithms*: LeCaR [18] and CACHEUS [15].
- *Object Learning Algorithms*: LRB [16] and 3L-Cache [25].

Table 1. Description of traces

Datasets	Object(10^6)	Request(10^6)	Type	Traces	Description
MSR	242	1355	I/O	5	I/O traces in an enterprise data center for one week
SQL Server	61	398	OLTP	8	SQL Server trace collected at Microsoft using the event tracing for Windows framework.
YCSB	356	4871	Key-Value	6	Key-Value trace widely utilized benchmark in numerous key-value stores

Metrics. We measure the performance of eviction algorithms by miss ratio and throughput. For the miss ratio, since the miss ratio varies in a wide range across traces, we present the miss ratio reduction compared to LRU: $\frac{MR_{LRU} - MR_{algo}}{MR_{LRU}} \times 100\%$, where MR stands for miss ratio. If an algorithm has a higher miss ratio than LRU, we present LRU's reduction in miss ratio relative to the algorithm, rather than bounding it between -100% and 100%. For throughput, we measure the throughput of an algorithm relative to LRU: $\frac{Throughput_{algo}}{Throughput_{LRU}}$.

Implementation. We implemented SL-Cache and the state-of-the-art eviction algorithms in libCacheSim [1]. All experiments are conducted on a server with two Intel(R) Xeon(R) Gold 6348 CPUs at 2.60 GHz, each with 28 cores and 2 threads per core, 256 GB of DRAM, and a 4 TB WD_BLACK AN1500 SSD. The server runs 64-bit Ubuntu 22.04.4 LTS. The file system is ext4.

4.2 Miss Ratio

In this section, we compare the miss ratio of different cache eviction algorithms. We evaluate cache sizes ranging from small to large across all workloads mentioned above. Figure 3 presents part of these results with miss ratio comparison results.

SL-Cache demonstrates improvements in miss ratio relative to other state-of-the-art algorithms across all workloads, for both small and large cache sizes.

For a small cache size, SL-Cache achieves the lowest miss ratio for 42.9% of the traces. For the closest competitors, LRB achieves the lowest miss ratio for 21.5% of the traces. On average, SL-Cache reduces the miss ratio by 13.61% across all traces compared to LRU. For other competitors, 3L-Cache, S3FIFO, LRB, TinyLFU, CACHEUS, LHD, Hill-Cache, SIEVE, LeCaR, and GL-Cache reduce the miss ratio by 12.59%, 11.94%, 11.82%, 11.30%, 8.66%, 8.51%, 6.67%, 3.69%, and 0.02%, respectively, on average across all traces compared to LRU. For workloads with small cache sizes, Hill-Cache outperforms object-level learning algorithms because it leverages hill-climbing to balance recency and frequency and triggers adjustments more frequently when the cache size is small.

For large cache sizes, SL-Cache achieves the lowest miss ratio across 73.6% of the traces. For the closest competitors, Hill-Cache achieves the lowest miss

ratio for 10.9% of the traces. On average, SL-Cache reduces the miss ratio by 10.20% across all traces compared with LRU. For other competitors, 3L-Cache, S3FIFO, LRB, TinyLFU, CACHEUS, LHD, Hill-Cache, SIEVE, LeCaR, and GL-Cache reduce the miss ratio by 9.07%, 8.49%, 6.37%, 7.11%, 5.68%, 1.94%, 6.68%, 0.73%, and 0.35%, respectively, on average across all traces compared to LRU. For some workloads with large cache sizes, S3FIFO is the most competitive heuristic, as it uses rapid demotion via a small FIFO filter to retain more hot objects.

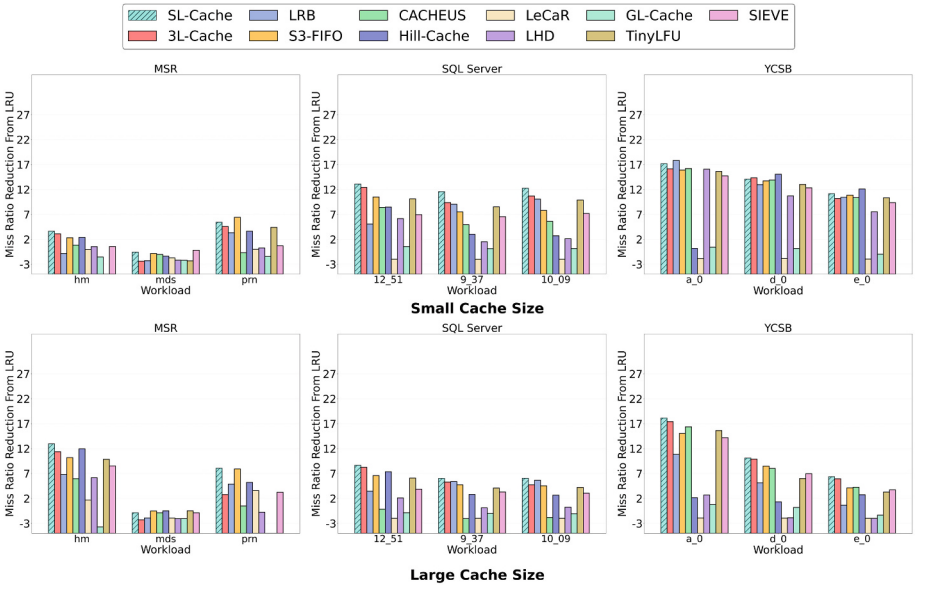

Fig. 3. Miss ratio reduction (**higher is better**) of different cache eviction algorithms with different cache sizes across workloads.

It is worth noting that LHD, LeCaR, CACHEUS, and GL-Cache cannot outperform state-of-the-art heuristic algorithms with quick demotion techniques such as S3FIFO, TinyLFU, and Hill-Cache across a large portion of workloads. This is because their coarse-grained predictions and reliance on specific heuristic features may not capture access patterns well enough to improve performance.

4.3 Throughput

Computational efficiency is a critical determinant for the practical deployment of learning-based eviction algorithms in high-performance systems. To provide a comprehensive evaluation, we measure the throughput of various algorithms relative to the LRU baseline across all workloads.

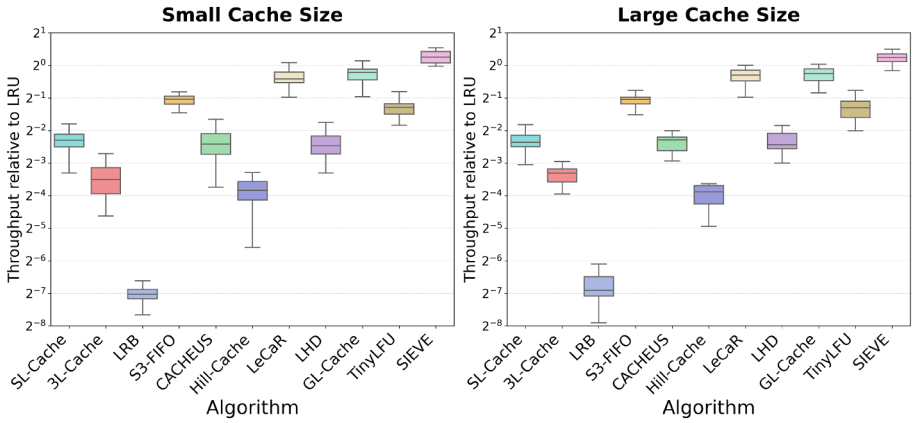

Fig. 4. Throughput comparison of different cache eviction algorithms with varying cache sizes across workloads.

Figure 4 illustrates the throughput distribution using boxplots for both small and large cache sizes. Our results demonstrate that SL-Cache successfully bridges the throughput gap between sophisticated object-level learning and simple heuristics.

At a small cache size, SL-Cache achieves a throughput significantly higher than that of existing object-level learning models. Specifically, SL-Cache achieves an average throughput approximately 2.3× higher than 3L-Cache and nearly 20× higher than LRB. Remarkably, by bypassing redundant model inferences for one-hit wonders, SL-Cache increases its throughput to a level comparable to coarse-grained learning algorithms such as CACHEUS and LHD, despite operating at a much finer granularity.

At a large cache size, SL-Cache's performance advantage remains robust. While the throughput of LRB drops drastically to approximately 2^{-7} relative to LRU, SL-Cache maintains a practical performance profile. Furthermore, the narrow range of SL-Cache's boxplots across diverse traces indicates its high predictability and robustness against varying access patterns.

4.4 Ablation Study

In this section, we analyze why SL-Cache reduces both latency and miss ratio by examining the improvements in the techniques employed. We analyze the performance improvement primarily in two areas: one-hit prediction bypass and a hot-oriented sample algorithm.

One-Hit Prediction Bypass. This technique uses pre-computed approximation results for one-hit object prediction and avoids heavy computation on one-hit object prediction to reduce computation overhead. We compare the prediction ratio across the entire trace of the object-level learning algorithms, including

SL-Cache, 3L-Cache, and LRB. We analyze the prediction ratio for small and large caches during trace execution in MSR and measure it online. LRB statically samples 64 objects per eviction, and the prediction ratio is fixed at 64. SL-Cache set the prediction ratio to 2 for both the sample and eviction algorithms. However, the prediction ratio may change if all sample objects are requested after the sample process. Although HALP is not applicable to our experiments, we note that it produces 3 rankwise predictions per eviction, with a prediction ratio of 3. For SL-Cache, because one-hit objects are bypassed during model prediction, their eviction does not require prediction. We set the prediction ratio for objects rather than for one-hit; hence, the total prediction ratio may be any value less than 2, depending on the one-hit object eviction rate in the workload's access pattern. As shown in Fig. 5, the prediction ratio of SL-Cache is constantly less than 2 and does not exceed 1 for most periods of the trace request for both small cache size and large cache size. As the most competitive algorithm with the lowest prediction ratio, 3L-Cache primarily maintains a prediction ratio of 2, which remains 100× higher than SL-Cache's maximum, underscoring SL-Cache's priority in reducing prediction.

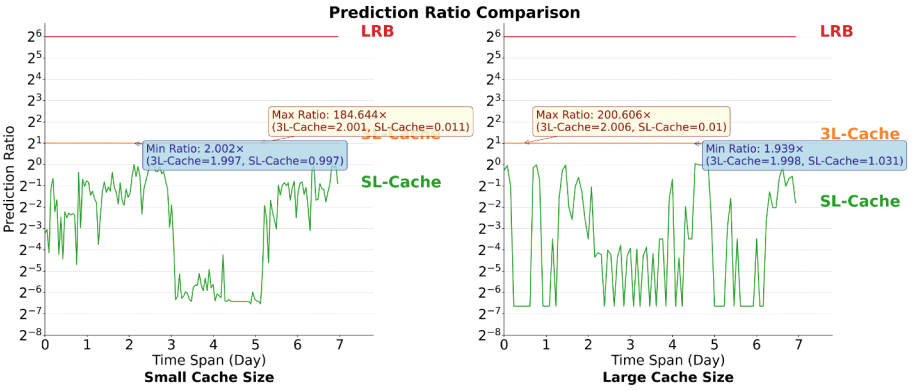

Fig. 5. Prediction ratio result for object-level learning algorithm during a trace in MSR.

Hot-Oriented Sampling Strategy. This technique filters out hot objects from the sample and detects cold objects with quick demotion. We design various sampling strategies to evaluate the performance of SL-Cache. (1) SL-Cache-clock is a variant of SL-Cache which uses 1 1-bit clock rather than a hit count to quickly demote hot objects. (2) SL-Cache-4 is another variant that sets the limit of hit count to 4 to test the necessity of high-frequency design for hit count. (3) SL-Cache-sd fixes the portion of a list in the cache structure and promotes all objects accessed more than once, which verifies the structure adjustment of SL-Cache.

As shown in Table 2, SL-Cache achieves a lower miss ratio compared to the other variants of SL-Cache. Although the variants achieve nearly the same miss ratio as SL-Cache, they fail to compete with SL-Cache across all workloads. This is because SL-Cache leverages the most comprehensive features to detect object hotness and adapt quickly to workload changes, providing high-quality sample results for prediction and eviction. Hence, SL-Cache outperforms other competitors across all workloads.

Table 2. Comparison of different sample strategies of SL-Cache.

Algorithm	MSR	SQLSever	YCSB	Total
SL-Cache	**9.1846**	**13.4731**	**17.4726**	**13.3768**
SL-Cache-clock	7.9142	11.5297	15.7145	11.7195
SL-Cache-sd	8.5914	13.4648	17.4568	13.1777
SL-Cache-4	8.2855	13.4070	16.9910	12.8945
Large Cache Size				
Algorithm	MSR	SQLSever	YCSB	Total
SL-Cache	**6.3961**	**7.8858**	**16.4511**	**10.2443**
SL-Cache-clock	3.8295	6.7974	14.6193	8.4154
SL-Cache-sd	5.2162	7.9241	16.2944	9.8116
SL-Cache-4	4.5898	7.9161	15.7075	9.4045

5 Conclusions

In this paper, we presented SL-Cache, a novel object-level learning eviction algorithm grounded on two key insights: selective prediction and hotness-aware sampling. SL-Cache identifies opportunities to bypass predictions for a significant number of one-hit objects and underscores the critical role of hotness-oriented sampling in object-level learning. Extensive evaluation showed that SL-Cache achieved the lowest miss ratio across diverse workloads and the highest throughput among object-level learning algorithms, comparable to coarse-grained learning approaches.

Future work will focus on three aspects. First, we will investigate the feasibility of using approximate predictions to achieve higher throughput while maintaining miss-rate stability. Second, we will consider implementing SL-Cache on some open-source platforms, e.g., RocksDB [5], and using additional workloads to evaluate SL-Cache. Third, the current SL-Cache is designed to accelerate query performance for web objects, such as those in content delivery networks [17,24]. We will extend SL-Cache to a broader range of applications by adding effective policies for dirty-object eviction.

Acknowledgments. This paper is supported by the National Science Foundation of China (grant no. 62072419). Peiquan Jin is the corresponding author.

References

1. libcachesim: a high-performance cache simulation. https://github.com/1a1a11a/libCacheSim. Accessed 21 Aug 2025
2. Beckmann, N., Chen, H., Cidon, A.: LHD: Improving cache hit rate by maximizing hit density. In: NSDI, pp. 389–403 (2018)
3. Belady, L.A.: A study of replacement algorithms for a virtual-storage computer. IBM Syst. J. **5**(2), 78–101 (1966)
4. Berger, D.S.: Towards lightweight and robust machine learning for CDN caching. In: HotNet, pp. 134–140 (2018)
5. Dong, S., Kryczka, A., et al.: Evolution of development priorities in key-value stores serving large-scale applications: the RocksDB experience. In: FAST, pp. 33–49 (2021)
6. Einziger, G., Friedman, R., Manes, B.: TINYLFU: a highly efficient cache admission policy. ACM Trans. Storage **13**(4), 1–31 (2017)
7. Jiang, S., Chen, F., Zhang, X.: Clock-pro: an effective improvement of the clock replacement. In: ATC, pp. 323–336 (2005)
8. Jin, P., Ou, Y., Härder, T., Li, Z.: AD-LRU: an efficient buffer replacement algorithm for flash-based databases. Data Knowl. Eng. **72**, 83–102 (2012)
9. Johnson, T., Shasha, D.: 2Q: a low overhead high performance buffer management replacement algorithm. In: VLDB, p. 439–450 (1994)
10. Li, Y., Hu, H., Lei, C., et al.: Hill-cache: adaptive integration of recency and frequency in caching with hill-climbing. In: ICDE, pp. 3947–3960 (2024)
11. Li, Z., Jin, P., Su, X., Cui, K., Yue, L.: CCF-LRU: a new buffer replacement algorithm for flash memory. IEEE Trans. Consum. Electron. **55**(3), 1351–1359 (2009)
12. Megiddo, N., Modha, D.S.: ARC: a self-tuning, low overhead replacement cache. In: FAST (2003)
13. O'neil, E.J., O'neil, P.E., Weikum, G.: The LRU-K page replacement algorithm for database disk buffering. ACM SIGMOD Record **22**(2), 297–306 (1993)
14. Ou, Y., Härder, T., Jin, P.: CFDC: a flash-aware buffer management algorithm for database systems. In: ADBIS, pp. 435–449 (2010)
15. Rodriguez, L.V., Yusuf, F., Lyons, S., et al.: Learning cache replacement with CACHEUS. In: FAST, pp. 341–354 (2021)
16. Song, Z., Berger, D.S., Li, K., et al.: Learning relaxed belady for content distribution network caching. In: NSDI, pp. 529–544 (2020)
17. Song, Z., Chen, K., Sarda, N., et al.: HALP: Heuristic Aided Learned Preference eviction policy for YouTube content delivery network. In: NSDI, pp. 1149–1163 (2023)
18. Vietri, G., Rodriguez, L.V., Martinez, W.A., et al.: Driving cache replacement with ML-based LeCaR. In: HotStorage (2018)
19. Wang, X., Jin, P., Yuan, Y.: SEMU: concurrency-optimized high-performance cache management for key-value caches. IEEE Trans. Comput. **75**(2), 734–747 (2026)
20. Yang, J., Mao, Z., Yue, Y., et al.: GL-Cache: group-level learning for efficient and high-performance caching. In: FAST, pp. 115–134 (2023)

21. Yang, J., Zhang, Y., Qiu, Z., et al.: FIFO queues are all you need for cache eviction. In: SOSP, pp. 130–149 (2023)
22. Yuan, Y., Jin, P.: Learned buffer management: a new frontier: work-in-progress. In: CODES/ISSS, pp. 25–26 (2021)
23. Yuan, Y., Jin, P.: Learned buffer replacement for database systems. In: DSDE, pp. 18–25 (2022)
24. Zhang, Y., Yang, J., Yue, Y., et al.: SIEVE is simpler than LRU: an efficient turn-key eviction algorithm for web caches. In: NSDI, pp. 1229–1246 (2024)
25. Zhou, W., Niu, Z., Xiong, Y., et al.: 3L-CACHE: low overhead and precise learning-based eviction policy for caches. In: FAST, pp. 237–254 (2025)