COMPUTER SCIENCE

A Simulation-Based Comparison of Cache Replacement Policies under Different Workloads and Cache Sizes

Joy Uchechukwu Ukibe, Chiagozie Excellence Ezeumeanya

Department of Computer Science, Faculty of Physical Sciences, Nnamdi Azikiwe University, Awka, Anambra State, Nigeria

Published November 2026

Abstract

The effectiveness of a cache replacement rule depends on the pattern of requests it receives and on how much space is available to retain previously requested items. This study used a deterministic simulator to compare least recently used (LRU), first-in-first-out (FIFO), and least frequently used (LFU) under three synthetic request patterns: uniform-random, Zipfian-skewed, and scan-heavy/mixed. For each workload, ten independently seeded traces of 50,000 requests were tested at capacities of 100, 250, 500, and 1,000 entries. The outcome was the proportion of requests served from the cache, reported as a mean and standard deviation across seeds. At capacity 500, LFU recorded the highest mean hit rate on the Zipfian workload (79.96% ± 0.22 percentage points), followed by LRU (75.88% ± 0.18) and FIFO (71.99% ± 0.19). On uniform-random requests, the three methods were almost indistinguishable, with hit rates of approximately 4.95% to 4.98%. In the specified scan-heavy/mixed workload, hit rates ranged from 4.12% to 4.56%. Larger caches generally improved hit rates, particularly under Zipfian access. These findings describe the behaviour of the stated simulator and input generators; they do not establish which policy will perform best in a production system. Tests using public application traces or a real storage workload are needed before making operational recommendations.

Keywords: cache replacement; FIFO; LFU; LRU; workload simulation

1 Introduction

A cache keeps selected items in a faster, limited-capacity layer so that later requests can be served without retrieving the items from a slower layer. When the cache is full and a new item is requested, a replacement rule determines which resident item leaves. Hit rate is therefore shaped by the replacement rule, the size of the cache, and the pattern of requests. Earlier work on replacement algorithms demonstrates why policy performance should be considered alongside the workload rather than ranked in isolation [1], [2]. Caching is also one of several techniques through which storage systems trade space against access time, a relationship examined in a workload-aware review by some of the present authors [3].

Benchmarking studies likewise stress the need to describe workloads precisely. The Yahoo! Cloud Serving Benchmark provides a configurable framework for comparing data-serving systems under defined request patterns [4]. Research on adaptive caching also examines how recency, frequency, scans, and changes in popularity affect policy behaviour [5]. Such work supports controlled comparisons, but simulated request streams should not be treated as substitutes for production traces.

This paper examines a deliberately narrow question: how three familiar replacement rules behave when request locality and cache capacity are changed in a controlled simulation. The contribution is

the explicit description of the workload generators, repeated seeded runs, and provision of code and machine-readable outputs for inspection. The study does not introduce a new algorithm and does not claim that its synthetic traces reproduce the behaviour of a deployed storage service.

1.1 Aim and research questions

The study aimed to measure how LRU, FIFO, and LFU hit rates change across defined synthetic workloads and cache capacities.

- **RQ1:** How do the three policies compare under uniform, popularity-skewed, and scan-heavy/mixed request patterns?
- **RQ2:** What changes in hit rate occur as the cache capacity increases for each workload and policy?
- **RQ3:** How consistent are the results across independent random seeds, and what limits follow from using synthetic traces?

2 Background

LRU removes the cached item whose most recent request occurred furthest in the past. FIFO removes the item that has been resident for the longest time, without considering later hits. LFU removes an item with the smallest accumulated request count. In this implementation, ties at the minimum frequency are resolved by removing the least recently used item among the tied candidates. That tie-breaking rule is part of this specific implementation and should not be assumed to describe every LFU variant.

Replacement policies have been examined in object caches, file systems, databases, and processor memory. Prior studies include analytical comparisons and evaluations using application traces [6], [7], [8], [9]. Adaptive and learning-based approaches, including CACHEUS and deep-reinforcement-learning methods, also show that newer policies are often designed to respond to varied or changing workloads [5], [10], [11]. The present work is a controlled baseline rather than a production-system study: it varies request locality and cache size while leaving latency, throughput, write behaviour, and system overheads outside its scope.

3 Methodology

3.1 Experimental design

A Python simulator evaluated LRU, FIFO, and LFU using three workload generators, four cache capacities (100, 250, 500, and 1,000 entries), and ten independent seeds (8,612,026 to 8,612,035). Each generated trace contained 50,000 requests. The design produced 360 policy-workload-capacity-seed evaluations (3 × 3 × 4 × 10). For a given workload and seed, all three policies received the same request sequence, allowing a direct comparison under identical input. The supplied script records the experiment settings and writes the raw and aggregated outputs.

3.2 Synthetic workload definitions

Uniform-random workload: every request independently selected one identifier from a set of 10,000, with all identifiers having equal probability. This workload provides a reference case with little persistent popularity structure.

Zipfian workload: requests were drawn from 10,000 identifiers with probabilities proportional to $1/r^{1.15}$, where r denotes the identifier's rank. Consequently, high-ranked identifiers were requested more often than lower-ranked ones.

Scan-heavy/mixed workload: the generator alternated between sequential scan segments and requests drawn from a small popular set. With probability 0.70, it appended a segment of 250 consecutive identifiers starting at a randomly selected valid position; otherwise, it selected an identifier from a 50-item hot set. Generation continued until 50,000 requests had been produced, and any excess was truncated. This is an artificial stress pattern created for the experiment, not a captured application trace.

3.3 Policy implementation and metrics

Each policy started with an empty cache. A request was counted as a hit if its identifier was already resident; otherwise, it was counted as a miss and inserted, with an eviction when the cache had reached capacity. LRU tracked the order of recent use, FIFO tracked insertion order, and LFU tracked request counts with least-recently-used tie-breaking among items tied for the lowest count. Hit rate was calculated as hits divided by all requests; miss rate was one minus the hit rate. For each condition, results were summarized across the ten seeds using the mean and standard deviation.

3.4 Reproducibility and verification

The supplementary Python file specifies the generators, seeds, capacities, replacement rules, and output files. The run-level CSV contains one record per evaluation, while the summary CSV reports means and standard deviations grouped by workload, capacity, and policy. The figures were produced from the summary data. These files make the reported calculations reproducible; they do not independently validate the simulator against a production trace or a separately implemented system.

3.5 Choice of outcomes and interpretation

Cache hit rate was selected as the primary outcome because it directly describes the proportion of requests served by the cache without requiring a fetch from the next storage layer. Miss rate is its complement and therefore provides no independent information in this experiment. Hit rate should not be confused with latency: a cache hit may still incur processing overhead, and a miss penalty depends on the architecture and storage medium. This simulation did not assign time costs to hits or misses, so it cannot estimate response time or throughput.

3.6 Fairness of policy comparisons

A fair policy comparison requires the algorithms to receive the same request sequence for a given workload and seed. The experiment was designed around paired inputs in this way. Differences in hit rate can therefore be attributed to the replacement decisions made by the implemented policies under that trace, subject to correct implementation. The experiment does not compare the CPU cost or memory overhead of maintaining policy metadata. In a deployed system, these overheads can affect the practical value of a small improvement in hit rate.

3.7 Scope of inference

The random seeds provide repeated samples from the specified synthetic generators; they do not represent independent hospitals, databases, users, or physical devices. The standard deviations quantify variability among generated traces under the chosen parameters. They are not confidence

intervals for production performance and should not be used to claim statistical significance between algorithms without a prespecified inferential analysis. The results are descriptive and conditional on the implementation and workload definitions.

4 Results

Table 1 reports mean hit rates and standard deviations at capacities of 500 and 1,000 entries. For the Zipfian workload, LFU had the highest mean hit rate at both sizes. At capacity 500, its hit rate was 79.96% (SD 0.22 percentage points), compared with 75.88% for LRU and 71.99% for FIFO. At capacity 1,000, LFU reached 84.03%, LRU 81.82%, and FIFO 78.63%. This is consistent with cumulative frequency counts helping to retain popular items in a workload whose popularity distribution remains stable; it should not be interpreted as proof that LFU is generally superior.

The uniform-random workload produced similar results for all three policies, with mean hit rates between 4.95% and 4.98% at capacity 500. The scan-heavy/mixed generator also produced low hit rates, between 4.12% and 4.56% at that capacity. Its long runs of distinct sequential identifiers create limited opportunity for reuse, even though requests to a small hot set are interspersed. These results depend on the exact scan length, hot-set size, mixing probability, and cache capacity used here.

Table 1. Mean cache hit rate (%) and standard deviation in percentage points across ten seeds.

Workload	Cap.	LRU	FIFO	LFU
Uniform-random	500	4.95 ± 0.12	4.95 ± 0.12	4.98 ± 0.14
Uniform-random	1,000	9.88 ± 0.12	9.89 ± 0.12	9.87 ± 0.16
Zipfian-skewed	500	75.88 ± 0.18	71.99 ± 0.19	79.96 ± 0.22
Zipfian-skewed	1,000	81.82 ± 0.15	78.63 ± 0.15	84.03 ± 0.17
Scan-heavy/mixed	500	4.47 ± 0.70	4.55 ± 0.64	4.12 ± 1.17
Scan-heavy/mixed	1,000	9.46 ± 0.98	9.54 ± 1.11	9.23 ± 1.42

Note: Standard deviations describe variation among the ten seeded runs. The supplementary CSV files contain results for all four capacities.

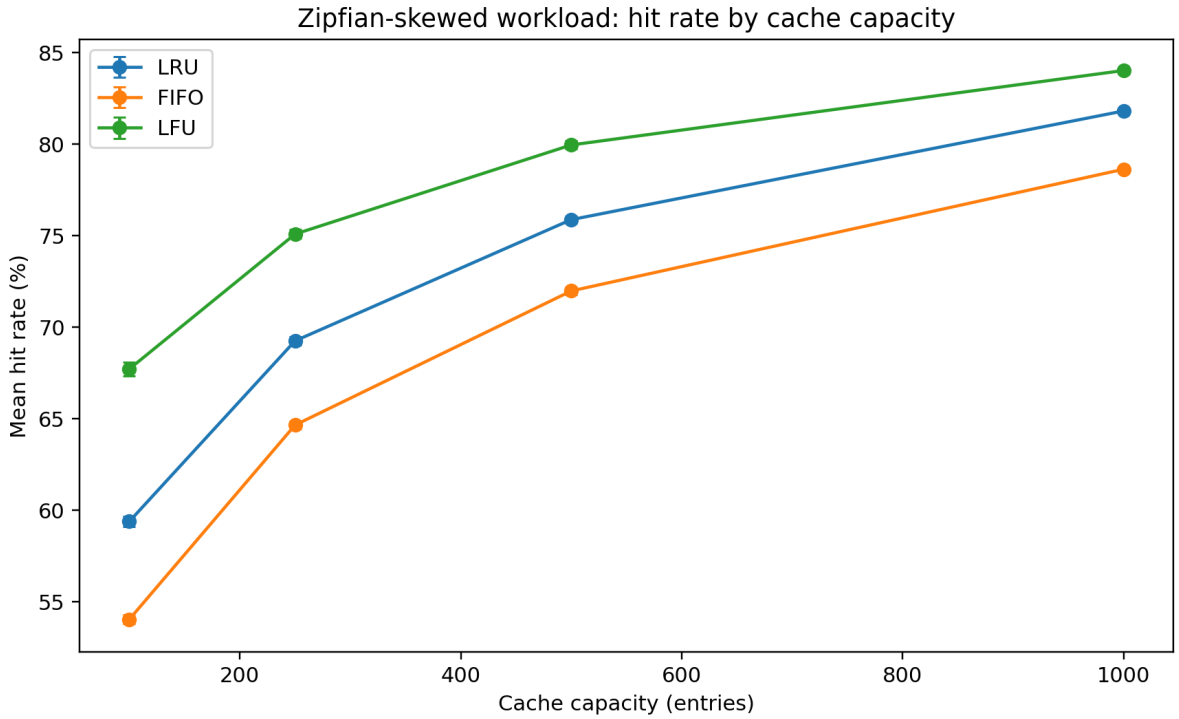


Figure 1. Mean hit rate across cache capacities for the Zipfian-skewed workload; error bars show one standard deviation across ten seeds.

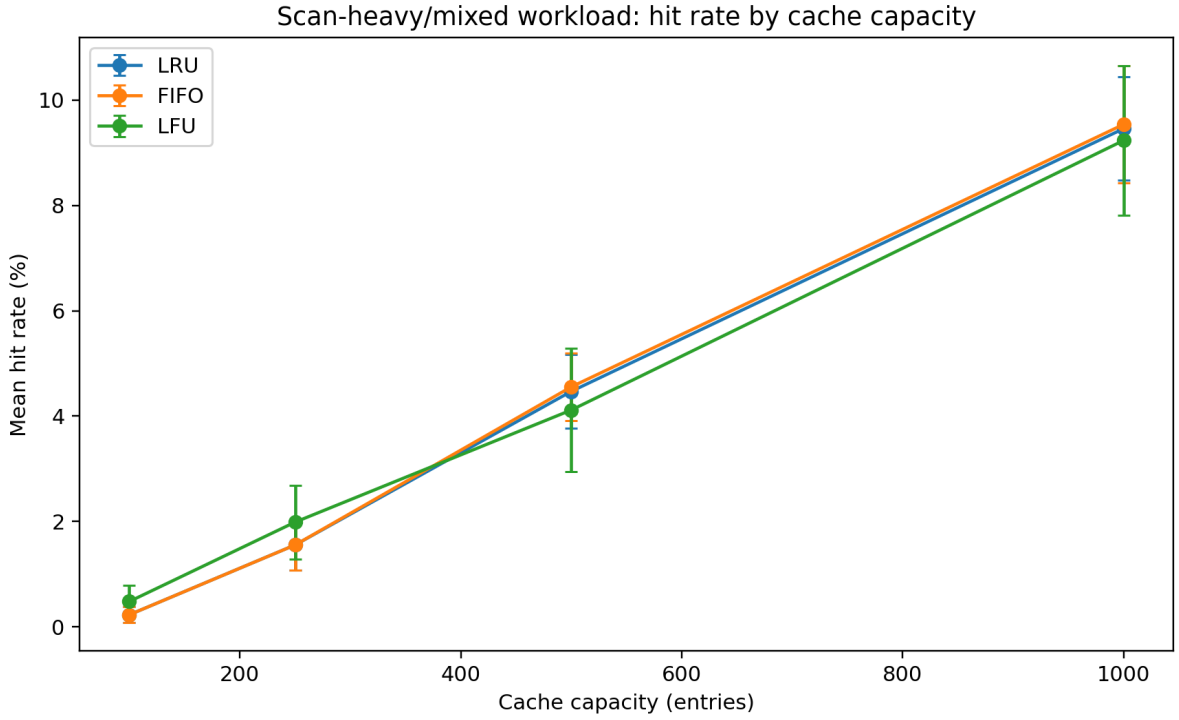


Figure 2. Mean hit rate across cache capacities for the scan-heavy/mixed workload; error bars show one standard deviation across ten seeds.

4.1 Results across all cache capacities

Table 2 reports the mean hit rates for all four capacities, and Figures 1 and 2 plot the Zipfian-skewed and scan-heavy/mixed results respectively.

The pattern was not restricted to the capacity-500 comparison. Across the Zipfian workload, mean hit rates rose as capacity increased from 100 to 1,000 entries for all three policies. LFU increased from 67.72% to 84.03%, LRU from 59.39% to 81.82%, and FIFO from 54.05% to 78.63%. The gap between policies narrowed as capacity increased, although LFU remained ahead under the stated stationary popularity distribution.

Table 2. Mean hit rate (%) by workload, cache capacity and replacement policy, shown as mean (SD) across ten seeded runs.

Workload	Cap.	LRU	FIFO	LFU
Uniform-random	100	1.00 (0.05)	1.00 (0.05)	1.01 (0.06)
Uniform-random	250	2.48 (0.09)	2.48 (0.08)	2.50 (0.08)
Uniform-random	500	4.95 (0.12)	4.95 (0.12)	4.98 (0.14)
Uniform-random	1,000	9.88 (0.12)	9.89 (0.12)	9.87 (0.16)
Zipfian-skewed	100	59.39 (0.27)	54.05 (0.25)	67.72 (0.38)
Zipfian-skewed	250	69.26 (0.22)	64.66 (0.21)	75.09 (0.25)
Zipfian-skewed	500	75.88 (0.18)	71.99 (0.19)	79.96 (0.22)
Zipfian-skewed	1,000	81.82 (0.15)	78.63 (0.15)	84.03 (0.17)
Scan-heavy/mixed	100	0.23 (0.15)	0.23 (0.15)	0.48 (0.31)
Scan-heavy/mixed	250	1.56 (0.47)	1.56 (0.47)	1.99 (0.70)
Scan-heavy/mixed	500	4.47 (0.70)	4.56 (0.64)	4.12 (1.17)
Scan-heavy/mixed	1,000	9.46 (0.98)	9.54 (1.11)	9.23 (1.42)

4.2 Variation between seeded runs

Variability differed by workload. Uniform-random runs showed relatively small standard deviations because the request probabilities were uniform and the traces were long. Zipfian runs also showed limited variation in the reported conditions, while the scan-heavy/mixed workload had greater variability, especially for LFU. This is consistent with the mixed generator's sensitivity to the randomly selected scan starting points and the timing of hot-set requests. The observed variation is descriptive; it does not establish that differences between policies are statistically significant.

5 Discussion

The results indicate that replacement-policy performance changes with workload locality. Under Zipfian access, a relatively small group of identifiers accounts for many requests. LFU benefits because repeated counts help retain these popular items; LRU also benefits from repeated access, whereas FIFO does not use request history when choosing a victim. When requests are uniformly distributed, neither recency nor frequency provides a strong advantage, which explains the similar hit rates observed in this condition.

The scan-heavy/mixed result should be interpreted in light of the generator used. The experiment combines long sequential segments with requests to a 50-item hot set, but other segment lengths, hot-set probabilities, or phase orderings could produce different results and possibly change the

policy ranking. The low hit rates therefore characterize this particular synthetic workload; they are not a general estimate for all scan-heavy applications.

The broader lesson is that caching strategies should be assessed against clearly specified workloads. The results do not demonstrate that LFU is the best choice for a real database or storage platform. Actual performance may also depend on object size, cache admission and invalidation, changing popularity, shared tenancy, memory and CPU overhead, read/write balance, and end-to-end latency. The experiment did not test more advanced policies such as ARC, LIRS, CLOCK-Pro, or learned replacement approaches.

5.1 Interpretation for cache design

The results support evaluating a cache policy in relation to the access pattern it is expected to serve. Where a workload has stable and highly skewed popularity, a frequency-based rule can retain repeatedly requested items even if they have not been accessed very recently. Recency-based rules can be more responsive when recently used items are likely to be requested again. FIFO is simpler conceptually because it does not need to update a recency or frequency score on every hit, but simplicity alone does not guarantee the highest hit rate. These are explanatory interpretations of the simulated patterns, not deployment recommendations.

5.2 Why the uniform workload is a useful control

The uniform-random trace provides a reference condition in which no identifier has a persistent popularity advantage. In such a setting, frequency and recency histories offer little reliable information about which identifier will be requested next. The small differences among LRU, FIFO, and LFU in this condition are therefore plausible within the model. This control also illustrates why a single aggregate result can conceal workload dependence: a policy that performs well on a skewed distribution may offer little benefit when requests are nearly uniform.

5.3 Interpreting the scan-heavy/mixed workload

The scan-heavy/mixed workload is intentionally demanding because sequential runs introduce many distinct identifiers that may not be requested again soon. The hot set creates some reuse, but the balance between sequential segments and hot-set accesses limits how much each policy can exploit it. LFU performs differently at different capacities in this workload, and its standard deviation is higher in several conditions. One possible explanation is that accumulated counts and randomly positioned scans interact with the timing of hot-set requests. This explanation is consistent with the design but should be tested with controlled parameter sweeps rather than treated as a demonstrated causal mechanism.

5.4 Relationship to benchmarking practice

Benchmarking work such as YCSB emphasizes that workload composition is part of the experiment and must be specified for results to be interpretable [4]. The present simulation follows that principle at a small scale by declaring request distributions, trace length, cache capacities, seeds, and policy definitions. Unlike a complete database benchmark, however, it does not include a storage engine, concurrent clients, update operations, network delay, persistence, or end-to-end measurements. Its value is therefore methodological transparency rather than a claim to model a full database service.

5.5 Reproducibility and auditability

Providing source code and machine-readable outputs allows a reader to inspect the policy implementation and recompute summary values. Reproducibility depends on retaining the exact code version, Python and package versions, random seeds, and definitions of the generators. For a formal research archive, the authors should place the script, CSV files, and a README in a stable repository or supplementary-material record and cite the permanent identifier in the final paper. The current package provides the core files, but it should not be described as having a permanent public archive unless one is actually created.

5.6 Practical implications and next steps

The current results suggest several testable follow-up questions. First, changing the Zipf exponent would show how sensitive policy rankings are to the degree of popularity skew. Second, varying the scan probability, scan length, and hot-set size would clarify which features drive the mixed-workload outcomes. Third, a phase-changing workload could test whether LFU's cumulative counts retain stale popularity after request patterns change. Fourth, additional policies such as ARC, LIRS, CLOCK-Pro, and LFU with aging could provide a more informative baseline. Finally, trace-driven experiments should measure not only hit rate but also latency, throughput, memory use, and algorithmic overhead.

5.7 Suggested validation protocol for a follow-up study

A follow-up study should predefine its workloads and comparison metrics, use identical traces for all algorithms, include a warm-up policy where appropriate, and report both central estimates and uncertainty. It should test multiple independent traces or public workloads and document any preprocessing. Where possible, results should be compared with an independently implemented reference simulator or published baseline. If inferential tests are used, the analysis should account for paired outcomes from the same traces and report effect sizes alongside p-values. None of these additional validations is claimed to have been completed in the present study.

6 Limitations

First, every request trace was synthetic; no public production trace, application benchmark, or physical storage system was tested.

Second, the comparison covered only three basic policies. The LFU implementation used cumulative counts without aging, so it may respond poorly when an item's popularity changes over time.

Third, the scan-heavy/mixed generator represents one design choice. Different scan lengths, hot-set sizes, and mixing ratios may yield different outcomes.

Fourth, hit rate was the only performance metric. The study did not measure latency, throughput, runtime or memory overhead, energy consumption, or write amplification.

Finally, ten seeds offer a useful initial view of run-to-run variation but do not cover the full range of possible workload parameters. The package has not been independently peer-reviewed or validated against external traces.

7 Conclusion

This simulation compared LRU, FIFO, and LFU over three synthetic request patterns and four cache capacities, with ten independently seeded runs for each condition. LFU achieved the highest hit rate under the Zipfian workload; the policies were nearly tied under uniform-random access; and all recorded low hit rates under the particular scan-heavy/mixed pattern. Larger capacities generally increased hit rates. The findings support a limited conclusion: replacement-policy performance depends on the workload and capacity, so a ranking from one synthetic trace should not be generalized to all systems. Future work should use public traces, include additional algorithms, and measure latency and resource costs alongside hit rate.

Declarations

Competing interests: The authors declare no competing interests.

Funding: No specific grant from any funding agency in the public, commercial, or not-for-profit sectors was received for this work.

Author contributions: Joy Uchechukwu Ukibe and Chiagozie Excellence Ezeumeanya contributed equally to this work, including conceptualization, simulation design, analysis, and manuscript preparation.

Data and code availability: The synthetic simulation source code, raw run-level CSV, and aggregated results CSV accompany this manuscript as supplementary files. The request traces are generated procedurally by the supplied code and were not collected from human participants or a production system.

Ethics statement: This computational simulation used no human participants, identifiable personal data, or animals; ethics approval and participant consent were not applicable.

References

[1] Belady LA. A study of replacement algorithms for a virtual-storage computer. *IBM Syst J*. 1966;5(2):78–101. https://doi.org/10.1147/sj.52.0078

[2] Robinson JT, Devarakonda MV. Data cache management using frequency-based replacement. In: *Proc*. 1990 ACM SIGMETRICS Conf. Measurement and Modeling of Computer Systems; 1990. p. 134–142. https://doi.org/10.1145/98457.98523

[3] Ukibe JU, Atuorah N, Ike IF, Akawuku GI, Ezeumeanya CE, Amadiegwu EN. Storage techniques and the space-access time trade-off: A workload-aware review and decision framework. *Int J Latest Technol Eng*. 2026;15(9):2982–2997. https://doi.org/10.51583/IJLTEMAS.2026.150900150

[4] Cooper BF, Silberstein A, Tam E, Ramakrishnan R, Sears R. Benchmarking cloud serving systems with YCSB. In: Proc. 1st ACM Symp. Cloud Computing; 2010. p. 143–154. https://doi.org/10.1145/1807128.1807152

[5] Rodriguez LV, et al. Learning cache replacement with CACHEUS. In: Proc. 19th USENIX Conf. File and Storage Technologies (FAST 21); 2021. p. 341–354.

[6] Hosseini-Khayat S. Replacement algorithms for object caching. In: *Proc*. 1998 ACM Symp. Applied Computing; 1998. p. 90–97. https://doi.org/10.1145/330560.330653

[7] Jiang S, Zhang X. LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. *ACM SIGMETRICS Perform Eval Rev*. 2002;30(1):31–42. https://doi.org/10.1145/511399.511340

[8] Megiddo N, Modha DS. ARC: A self-tuning, low overhead replacement cache. In: *Proc*. 2nd USENIX Conf. File and Storage Technologies (FAST '03); 2003. p. 115–130.

[9] Park NE, Kim J, Jeong TS. Cache memory and replacement algorithm implementation and performance comparison. *J Korea Soc Comput Inf*. 2020;25(3):11–17. https://doi.org/10.9708/jksci.2020.25.03.011

[10] Zhou Y, Wang F, Shi Z, Feng D. An end-to-end automatic cache replacement policy using deep reinforcement learning. In: *Proc*. Int. Conf. Automated Planning and Scheduling; 2022. p. 537–545. https://doi.org/10.1609/icaps.v32i1.19840

[11] Zhou Y, Wang F, Shi Z, Feng D. An efficient deep reinforcement learning-based automatic cache replacement policy in cloud block storage systems. *IEEE Trans Comput*. 2024;73(1):164–177. https://doi.org/10.1109/TC.2023.3325625