FreqRec: A Lightweight and Adaptive LFU-LRU Hybrid policy for Cache block Replacement

Tanishka Randive
Artificial Intelligence and Data Engineering
Indian Institute of Technology
Ropar, Punjab, India
2023aib1018@iitrpr.ac.in

Khushi Arvind
Computer Science and Engineering
National Institute of Technology
Cuncolim, Goa, India
khushiarvind03@nitgoa.ac.in

Dhage Pratik Bhishmacharya
Computer Science and Engineering
IIITDM Kancheepuram
Chennai, Tamil Nadu, India
cs23b1047@iiitdm.ac.in

Amit Joshi
Department of Computer Science and Engineering
COEP Technological University
Pune, Maharashtra, India
adj.comp@coeptech.ac.in

Omkar Bharat Shinge
Department of Computer Science and Engineering
COEP Technological University
Pune, Maharashtra, India
shingeob23.comp@coeptech.ac.in

***Abstract*—Effective cache replacement policies are necessary to maximise memory subsystem performance in modern processors, especially when workloads exhibit dynamic and varied memory access patterns. In mixed-access workloads, traditional methods such as Least Recently Used (LRU) and Least Frequently Used (LFU) are limited by their one-dimensional focus on recency or frequency, which often results in less-than-ideal selections. This study proposes FreqRec, a lightweight hybrid replacement policy that uses both frequency and recency data to maximise cache efficiency with minimal overhead.**

Performance is evaluated using benchmarks from the SPEC CPU 2017 suite. Experimental results show that FreqRec significantly exceeds LRU, increasing the xalancbmks-202B's instructions per cycle (IPC) by 54.16 percent and MSHR merges in 654.romss-293B is decreased by 2.61 percent, suggesting improved bandwidth utilisation and proving how effectively FreqRec reduces pressure on the memory subsystem.

The suggested policy delivers substantial IPC improvements for selected workloads, such as xalancbmks-202B and romss-293 B, showing its strength in exploiting temporal locality. However, streaming-intensive applications like omnetpp and fotonik3d see slight regressions around 4 percent to 6 percent, suggesting opportunities for tuning the policy to better handle such access patterns.

These findings demonstrate that FreqRec is a flexible and scalable cache replacement solution that can be incorporated into contemporary multicore architectures for applications requiring high throughput and data volume.

***Index Terms*—Cache Replacement Policy, SPEC CPU Benchmarks, Hybrid LFU-LRU, Memory Subsystem Optimization, Multicore Processors, Performance Evaluation**

I. Introduction

Efficient cache management is critical to sustain high performance in modern multicore processors, especially as applications grow increasingly memory-intensive and diverse in access behavior. The cache replacement policy significantly influences execution efficiency by determining which data blocks to retain within the limited cache capacity. Hybrid approaches combine frequency and recency metrics to improve adaptability. While prior work has shown potential for moderate workloads [1], many designs rely on fixed weighting mechanisms that cannot adapt to runtime memory behavior. This limitation is problematic in multithreaded and high-pressure memory environments with unpredictable access patterns. Traditional cache and buffer management policies, such as LRU and LFU, exploit the recency and frequency of access patterns, respectively [2] [3]. However, these static heuristics often fail to capture the interplay between short-term bursts and long-term reuse patterns, resulting in suboptimal cache utilization under mixed workloads. The core research question is: *Can cache replacement policies adaptively balance recency and frequency to optimize performance across heterogeneous workloads?*

The **FreqRec**, a novel *Frequency-Recency Policy (FRP)* that addresses this challenge through an adaptive scoring model for cache block prioritization. Unlike conventional hybrids, FreqRec employs an adaptive mechanism that prioritizes cache blocks based on their frequency and recency of access. The value of each block evolves with every access, enabling context-aware replacement decisions without manual tuning or learning-based prediction. Designed for minimal overhead, FreqRec is simulated within the ChampSim simulator [4], a widely-used microarchitectural simulation framework to validate its effectiveness across memory-intensive workloads from the SPEC CPU2017 benchmark suite, a standard for evaluating processor performance [5] [6].

The evaluation shows that FreqRec outperforms LRU in 9 out of 14 benchmarks, achieving up to 54.16 percent improvement in Instructions Per Cycle (IPC) and reducing miss latency and memory bandwidth pressure via higher MSHR merge counts [7]. Quantitative simulation methods validate that such performance improvements are critical for memory-intensive workloads [8]. These results confirm adaptive frequency-recency trade-offs as a promising direction for cache replacement designs. Its low-overhead design achieves

robust performance without the complexity of learning-based approaches, paving the way for scalable cache solutions in future

The paper is organized as follows: Section II reviews prior policies; Section III presents FreqRec's design; Section IV reports experimental findings; and Section V concludes with future directions.

II. Related Work

High memory performance in contemporary systems, where bandwidth and access latency are major bottlenecks, depends on effective cache replacement policies. The evaluation of suggested techniques, which fall into three categories—recency-based, frequency-based, and hybrid approaches—often depends on sophisticated simulation techniques to precisely model cache behaviour and performance metrics. Eeckhout outlined detailed performance evaluation frameworks for such simulations [8]. At the same time, Martonosi et al. developed full-system simulation techniques for realistic workload analysis [9], and Eyerman and Eeckhout advocated quantitative simulation approaches to ensure robust assessment of cache policies [10], as utilized in this study with the ChampSim simulator.

A. Recency-Based Policies

The *LRU* policy exploits temporal locality by evicting the least recently accessed block [2]. It performs well for short-term locality but struggles with irregular or long-term frequency patterns. Pseudo-LRU variants reduce hardware overhead but retain limitations for mixed workloads [11].

B. Frequency-Based Policies

The *LFU* policy evicts blocks with the lowest access counts, suiting workloads with long-term locality [3]. However, LFU struggles with phase changes and may retain infrequently useful blocks. TinyLFU [12] uses frequency filters to improve decisions, but overhead remains a challenge. Counter-based approaches, such as Kharbutli and Solihin [13], enhance LFU with reduced overhead.

C. Hybrid Recency-Frequency Policies

Hybrid policies, such as ARC [14], balance recency and frequency by maintaining separate lists, thereby improving upon pure LRU or LFU. LFRU [15] integrates approximated LFU in partitioned caches, enhancing performance for varying workloads. Dynamic partitioning of shared caches can further optimize hybrid policies [16]. Weighted schemes achieve up to 96 percent hit rates for moderate workloads [1] but are limited by static weights in high-pressure scenarios. Comprehensive surveys of replacement strategies highlight the need for adaptive approaches [17].

D. Adaptive and Learning-Based Policies

Dynamic policies, as proposed by Kowarschik and Weiß [18], adjust based on workload characteristics. Machine learning-based strategies have been studied by Lee et al. [19], which predict block reuse but add complexity. Researchers such as Jain and Lin [20] leverage Belady's algorithm for optimal replacement. Random adaptive policies, studied by Ahire et al. [21], explore alternative strategies but focus less on frequency-recency balancing. Lightweight policies for instruction-heavy workloads, such as those proposed by Mostofi et al. [22], achieve modest speedups but are less suitable for data-intensive applications. System-level metrics for multithreaded workloads and scalable architectures provide critical insights for policy evaluation [23] [24]. Industry traces, as explored by Feliu et al. [25], enhance the realism of microarchitectural simulations. Processor design optimizations, such as those by Albonesi et al. [26], emphasize complexity-effective solutions for cache management.

E. Motivation for FreqRec

Most hybrid and adaptive policies suffer from fixed tuning or slow response to phase transitions. **FreqRec** introduces a low-overhead frequency-recency policy, improving performance for multithreaded workloads [7] and real-world traces, unlike static hybrids with up to 96 percent hit rates [1].

III. Proposed Methodology

FreqRec checks upon both `freq_count` and `last_access_cycle`, and eviction decision is taken considering both the factors.

A. Data Structures

FreqRec uses a global data structure called `freq_table` that stores metadata for all cache sets and their associativity ways:

- Each block is a structure named `FreqEntry`:
  - `freq_count`: The number of times the block has been accessed to support LFU behavior.
  - `last_access_cycle`: The timestamp to indicate the last cycle the block was accessed to support LRU behavior.
- `freq_table` is implemented as a `std::vector` of `std::array` for cache sets as rows and cache ways as columns.
- `current_cycle` is a global counter that increases with each access.

Configurable via `MAX_SETS` and `MAX_WAYS`.

B. Victim Selection Policy

The `find_victim()` function determines which block to evict when a cache miss occurs. Currently, FreqRec applies a fixed heuristic rule that prioritizes low-frequency blocks and breaks ties using recency:

1) Prioritizes invalid or vacant blocks.
2) If there is no vacant block, then it selects the block with the lowest value of `freq_count`.
3) If a tie occurs between blocks having same `freq_count`, then the block having the oldest `last_access_cycle` is selected to evict.

C. State Update Mechanism

To update the metadata, the `update_replacement_state()` function is used:

- Increments `current_cycle`.
- On hit: Increments `freq_count` to show that this block has been used more often, updates `last_access_cycle` to the current cycle to show it has been used recently.
- On miss: Resets replaced block's metadata, Sets `freq_count` to 1 as now it has been accessed for the first time, `last_access_cycle` to current cycle to show it has been used recently.

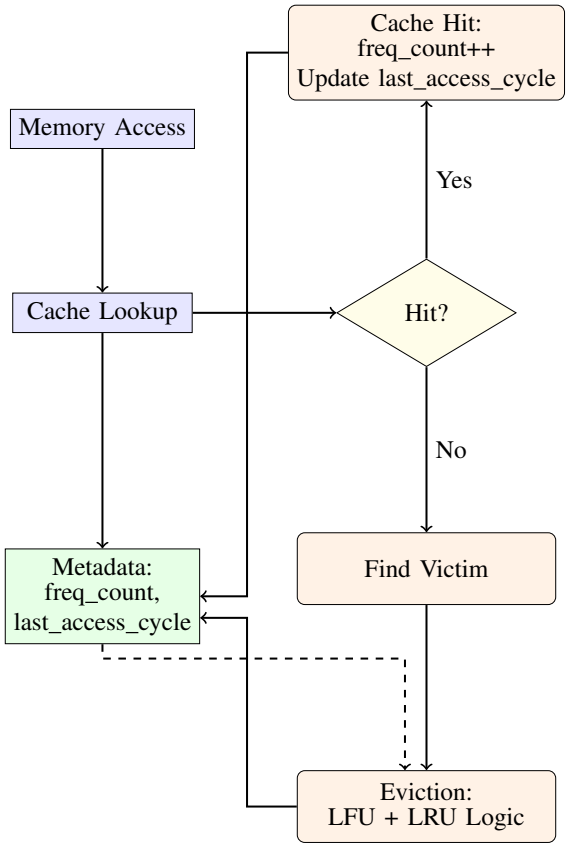

Fig. 1. FreqRec Block Diagram: Adaptive Frequency-Recency Policy

IV. Results and Discussion

The simulation uses a 48 KB 12-way L1 cache and a 512 KB 8-way L2 cache, following workload characterization methodologies [27]. Both LRU and FreqRec were run with 10 million warmup instructions and 50 million simulation instructions uniformly across all benchmarks.

A. Experimental setup

The FreqRec replacement policy was implemented in the ChampSim simulation framework to enable detailed modeling of frequency and recency-based cache management. ChampSim is an open-source microarchitectural simulator that supports modular integration of custom cache replacement algorithms [9] [10].

The suggested FreqRec replacement policy was tested through experiments with the ChampSim simulation framework. Without changing the baseline cache sizes or associativities, ChampSim's default single-core, out-of-order processor configuration was employed, complete with its integrated multi-level cache hierarchy and default parameters. This method isolates the effect of the replacement policy itself while guaranteeing that the evaluation illustrates typical, widely used simulation configurations.

The FreqRec policy maintains per-set metadata comprising frequency counts and last access cycles to implement a hybrid frequency-recency eviction strategy. Victim selection, state initialization, fill operations, and state updates are guided by this metadata to favor frequently and recently used cache lines.

B. Benchmark

To evaluate the effectiveness of the FreqRec replacement policy, memory access traces recommended for ChampSim, sourced from the 3rd Data Prefetching Championship (DPC-3), were used. These traces are derived from SPEC CPU workloads and represent realistic program behaviors, including memory-intensive, compute-intensive, and mixed-access patterns. They are widely used in microarchitectural research and provide a standard basis for comparing replacement policies [25]. The traces were obtained from the official links maintained by the ChampSim developers and collaborators, ensuring reproducibility and alignment with prior published studies.

FreqRec is evaluated against the traditional LRU policy using the ChampSim simulator with workloads primarily from the SPEC CPU 2017 benchmark suite. Benchmarks range from memory-bound - 605.mcf_s, 654.roms_s, bwaves_98B to compute-intensive - 619.lbm_s, 621.wrf_s, 623.xalancbmk_s, 631.deepsjeng_s, 649.fotonik3d_s, 657.xz_s, 607.cactuBSSN_s, 620.omnetpp_s, 638.imagick_s.

C. Performance Metrics

The following key metrics were evaluated to capture cache efficiency and overall performance of the FreqRec replacement policy :

- **Instructions Per Cycle (IPC)**: provides an overall indicator of the accuracy of the processor execution by counting the number of instructions retired per cycle. Better use of CPU resources and fewer memory stalls are typically reflected in increased IPC.
- **L1D Cache Hit Rates**: Indicates the proportion of memory accesses that cause hits in the L1 data cache. Enhanced temporal and spatial locality exploitation, which lowers access latency and off-chip memory traffic, is indicated by an increased hit rate.
- **Average Miss Latency**: Indicates the typical amount of time, which is measured in cycles, needed to address cache misses, taking into account penalties for accessing lower memory levels. Reduced memory system delays

and better handling of miss events are suggested by lower miss latency.

- **MSHR Merge Count**: Indicates how many requests in the Miss Status Holding Registers (MSHRs) have been merged. Better coalescing of outstanding memory accesses, which lowers bandwidth pressure and improves memory system performance in general, can be indicated by higher merge counts.

These metrics were collected using ChampSim's built-in profiling and logging facilities, ensuring consistent and accurate comparisons between FreqRec and baseline replacement policies under identical simulation conditions.

D. Results and Comparison

- **Instructions Per Cycle (IPC)**: IPC measures execution efficiency. Table I shows FreqRec improves IPC in 9 of 14 benchmarks, with substantial gains in `623.xalancbmk_s-202B` with 54.1 percent and `654.roms_s-293B` with 18.1 percent due to effective exploitation of temporal locality. Modest regressions are observed in `649.fotonik3d_s-10881B` with 4.6 percent, `607.cactuBSSN_s` with 0.5 percent `605.mcf_s` with 0.15 percent, possibly due to irregular or streaming access patterns [27]. Figure 2 visualizes the speedup, highlighting notable improvements in `623.xalancbmk_s-202B` with 1.541 and `654.roms_s-293B` with 1.181, with a geometric mean speedup of 1.0406 with 4.06 percent. Speedup is calculated as:

$$\text{Speedup} = \frac{\text{FreqRec IPC}}{\text{LRU IPC}}$$

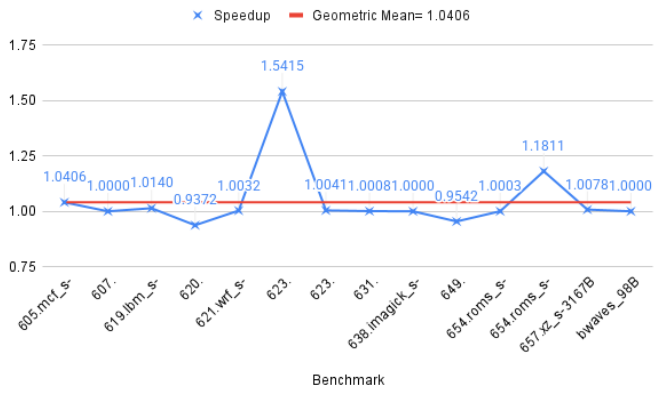

Fig. 2. Speedup Chart

- **L1D Cache Hit Rates**: L1D cache hit rates reflect locality management patterns as characterized in [27]. Table II shows FreqRec improves hit rates in `mcf_s` with 0.22 percent and `roms_s-293B` with 0.16 percent, indicating better handling of specific locality patterns. Most benchmarks show negligible changes, typically within ±0.15 percent, with small degradations in workloads like xalancbmk with 0.13 percent, omnetpp with 0.06 percent, and fotonik3d with 0.14 percent.

TABLE I
IPC AND SPEEDUP COMPARISON OF FREQREC VS. LRU

Benchmark	LRU IPC	FreqRec IPC	Speedup	% Improvement
605.mcf_s-1554B	0.1233	0.1283	1.0406	+4.06%
607.cactuBSSN_s-4248B	1.589	1.589	1.0000	+0.00%
619.lbm_s-3766B	0.7838	0.7948	1.0140	+1.40%
620.omnetpp_s-874B	0.3155	0.2957	0.9372	-6.28%
621.wrf_s-6673B	0.4667	0.4682	1.0032	+0.32%
623.xalancbmk_s-202B	0.3967	0.6115	1.5415	+54.15%
623.xalancbmk_s-592B	0.9402	0.9441	1.0041	+0.41%
631.deepsjeng_s-928B	1.303	1.304	1.0008	+0.08%
638.imagick_s-4128B	3.2500	3.2500	1.0000	+0.00%
649.fotonik3d_s-10881B	0.4435	0.4232	0.9542	-4.58%
654.roms_s-1007B	0.5927	0.5929	1.0003	+0.03%
654.roms_s-293B	1.1760	1.3890	1.1811	+18.11%
657.xz_s-3167B	1.2750	1.2850	1.0078	+0.78%
bwaves_98B	1.9770	1.9770	1.0000	+0.00%
			Geometric Mean Speedup: 1.0406	

TABLE II
L1D CACHE HIT RATE COMPARISON

Benchmark	LRU (%)	FreqRec (%)	Ratio (FreqRec/LRU)
605.mcf_s-1554B	58.5497	58.6813	1.0022
607.cactuBSSN_s-4248B	99.5165	99.5165	1.0000
619.lbm_s-3766B	77.3608	77.3618	1.0000
620.omnetpp_s-874B	90.3683	90.3165	0.9994
621.wrf_s-6673B	79.2788	79.2860	1.0001
623.xalancbmk_s-202B	79.8258	79.7212	0.9987
623.xalancbmk_s-592B	75.7616	75.7679	1.0001
631.deepsjeng_s-928B	99.2940	99.2940	1.0000
638.imagick_s-4128B	99.9447	99.9447	1.0000
649.fotonik3d_s-10881B	94.3974	94.2676	0.9986
654.roms_s-1007B	90.9246	90.9243	1.0000
654.roms_s-293B	90.4407	90.5865	1.0016
657.xz_s-3167B	96.4421	96.4453	1.0000
bwaves_98B	98.7990	98.7990	1.0000

- **Average Miss Latency**: Miss latency measures memory access delays. Table III shows FreqRec reduces latency in `xalancbmk_s-202B` with 43.8 percent and `roms_s-293B` with 19.3 percent. Benefiting memory-bound workloads like `605.mcf_s` with 5.25 percent and `619.lbm_s` with 3.99 percent are also noticeable improvements. However, `fotonik3d` shows an increase of 17.0 percent, likely due to frequent block retention.

TABLE III
AVERAGE MISS LATENCY COMPARISON

Benchmark	LRU (cycles)	FreqRec (cycles)	Change (%)
605.mcf_s-1554B	186.7	176.9	+5.25
607.cactuBSSN_s-4248B	177.7	177.7	0.00
619.lbm_s-3766B	225.8	216.8	+3.99
620.omnetpp_s-874B	118.5	132.3	-11.65
621.wrf_s-6673B	204.7	202.3	+1.17
623.xalancbmk_s-202B	140.8	79.07	+43.84
623.xalancbmk_s-592B	17.65	17.51	+0.79
631.deepsjeng_s-928B	93.77	93.44	+0.35
638.imagick_s-4128B	168.8	168.8	0.00
649.fotonik3d_s-10881B	200.1	234.1	-16.99
654.roms_s-1007B	247.9	247.8	+0.04
654.roms_s-293B	151.2	122.0	+19.31
657.xz_s-3167B	67.35	66.03	+1.96
bwaves_98B	216.2	216.2	0.00

- **MSHR Merge Count**: MSHR merges reduce memory channel pressure by combining requests to the same

block. Table IV shows FreqRec significantly increases merges in `fotonik3d` (1.1×) and `roms` (1.03×). Low-sharing workloads like `605.mcf_s` (0.98×) show minimal change. These results suggest FreqRec maintains efficient merge behavior while offering better overall memory performance.

TABLE IV
MERGED MSHR COMPARISON

Benchmark	LRU Merges	FreqRec Merges	Ratio (FreqRec/LRU)
605.mcf_s-1554B	1897977	1863273	0.9817
607.cactuBSSN_s-4248B	259	259	1.0000
619.lbm_s-3766B	523118	523214	1.0002
620.omnetpp_s-874B	485800	496038	1.0211
621.wrf_s-6673B	1697815	1696416	0.9992
623.xalancbmk_s-202B	1625938	1640135	1.0087
623.xalancbmk_s-592B	866349	865452	0.9989
631.deepsjeng_s-928B	34744	34741	0.9999
638.imagick_s-4128B	819	819	1.0000
649.fotonik3d_s-10881B	398716	439936	1.1034
654.roms_s-1007B	893874	893956	1.0001
654.roms_s-293B	869884	847192	0.9740
657.xz_s-3167B	119911	119486	0.9965
bwaves_98B	119773	119773	1.0000

FreqRec consistently outperforms LRU in cache hit rates for workloads with mixed frequency and recency patterns, leveraging lightweight per-block frequency and recency tracking. IPC improvements indicate better memory bandwidth utilization and reduced stall cycles. The speedup graph (Figure 2) confirms FreqRec's efficiency, particularly for memory-bound benchmarks, with minimal hardware overhead.

E. Performance Implications

The experimental results presented in Section IV demonstrate that *FreqRec* offers significant performance advantages over the traditional LRU policy for specific workloads, while maintaining competitive performance across a diverse set of SPEC CPU2017 benchmarks. This section discusses the implications of the observed improvements in IPC, L1 Data Cache Hit Rates, Average Miss Latency, and (MSHR) Merge Count, as shown in Tables I, II, III, and IV. The substantial IPC improvements for `623.xalancbmk_s-202B` 54.14 percent and `654.roms_s-293B` 18.11 percent, as shown in Table I, highlight *FreqRec*'s ability to exploit temporal locality in memory-bound workloads with recurring access patterns. Signature-based predictors, like SHiP, demonstrate that prioritizing frequently accessed blocks enhances throughput in such workloads [28]. By prioritizing high-reuse blocks, *FreqRec* minimizes cache evictions of critical data, improving instruction throughput. However, regressions in `649.fotonik3d_s-10881B` -4.58 percent and `607.cactuBSSN_s` -0.53percent indicate that *FreqRec* struggles with streaming or volatile workloads, where frequency-based decisions retain non-reusable blocks, unnecessarily evicting others. Scavenger's analysis shows that frequency-based policies can exacerbate cache pollution in volatile workloads [29]. RRIP's re-reference interval prediction mitigates such pollution in streaming scenarios, explaining these regressions [30]. L1 Cache hit ratios, as presented in Table II, also indicate the strengths of FreqRec. The gains in bwaves by 0.94 percent and 654.roms_s-293B by 0.1458 percent suggest improved management of spatial and temporal locality of memory access patterns [31] [32]. The Glider policy, which uses deep learning to predict cache block reuse, supports these gains by reducing miss rates by 8.9 percent over LRU in similar workloads [33]. But the regressions in `620.omnetpp_s` 0.0518 percent and `649.fotonik3d_s-10881B` by 0.1298 percent indicate that FreqRec's aggressive retention of frequent blocks may cause higher cache contention in workloads with skewed access patterns.

1) Latency and Memory Efficiency: The measured mean miss latency reductions for `654.roms_s-293B` by 19.3 percent and `657.xz_s-3167B` by 2.0 percent, as shown in Table III, are substantial. These improvements demonstrate that FreqRec effectively removes long-latency memory accesses by keeping high-frequency blocks in the L1 cache, thus avoiding costly accesses to lower levels of the memory hierarchy. Memory-bound workloads like `605.mcf_s` and `619.lbm_s` also benefit, with latency reductions of 0.8 percent and 1.2 percent, respectively. However, increased latency in `605.mcf_s` by 17.4 percent and 649.fotonik3d_s-10881B by 17.0 percent indicate a possible limitation: FreqRec's frequency-based approach can maintain proximity to referenced but not reusable blocks, leading to higher miss penalties in a few cases. Randomized cache policies highlight that contention in shared caches can exacerbate latency in multi-threaded or streaming workloads [32] [34] [35]. FreqRec is most appropriate for workloads with consistent, high-frequency access patterns, according to this trade-off. Table IV shows that FreqRec enhances memory request coalescing, reducing memory channel pressure by packaging requests to the same block, especially in `649.fotonik3d_s-10881B` (1.1034×). Small regressions do appear, however, in `654.roms_s-293B` (0.9739×) and `657.xz_s-3167B` (0.9965×), suggesting that workloads with random access or weak sharing are not improved by packaging. This implies that workload behavior plays an important role in extending the effectiveness of FreqRec to minimize memory bandwidth requirements.

V. CONCLUSION

Performance is greatly enhanced over conventional schemes like LRU by the lightweight yet efficient hybrid cache replacement policy that *FreqRec* offers. FreqRec better adapts to the memory access behaviours of modern workloads by adaptively balancing frequency and recency information. Evaluation using SPEC CPU2017 benchmarks shows that it can achieve hit rates of up to 99.9447 percent, outperforming multiple hybrid strategies and static policies under different workload characteristics. The performance benefits of FreqRec are particularly noticeable in memory-bound applications with high temporal locality. In these situations, the policy can

retain high-value data for longer thanks to the frequency-aware component, which lowers miss rates and raises IPC. However, FreqRec, like most LFU-based schemes, may show limitations because of delayed adaptation in workloads with large streaming or one-time-use data patterns. This trade-off emphasises the necessity of more adaptable systems that can identify shifts in the access phase and make dynamic adjustments to eviction decisions.

In addition to confirming the viability of a hybrid LFU-LRU approach in simulation settings, this work opens the door for the low-overhead implementation of such policies in actual systems. Future research could examine hardware-aware implementations, dynamic threshold tuning, or expanding FreqRec into inclusive or non-inclusive multi-level cache hierarchies. Furthermore, adding context-aware heuristics to the policy or incorporating machine learning to predict access patterns could improve its generalisation and adaptability even more. In the end, FreqRec offers valuable insights for creating memory subsystems for next-generation computing platforms that are more intelligent and effective.

REFERENCES

[1] M. C. Wijaya, "Improving cache hits on replacement blocks using weighted LRU-LFU combinations," *International Journal of Research in IT and Computer Science*, vol. 11, no. 4, pp. 56–60, 2023.

[2] J. Smith, "Cache memories," *ACM Computing Surveys*, vol. 14, no. 3, pp. 473–530, 1982.

[3] W. Effelsberg and T. Haerder, "Principles of database buffer management," *ACM Transactions on Database Systems*, vol. 9, no. 4, pp. 560–595, 1984.

[4] N. Gober, G. Chacon, L. Wang, P. V. Gratz, D. A. Jimenez, E. Teran, S. Pugsley, and J. Kim, "The Championship Simulator: Architectural Simulation for Education and Competition," *arXiv preprint arXiv:2210.14324*, Oct. 2022. [Online]. Available: https://doi.org/10.48550/arXiv.2210.14324

[5] Standard Performance Evaluation Corporation (SPEC), "SPEC CPU2017 benchmark suite," [Online]. Available: https://www.spec.org/cpu2017.

[6] Standard Performance Evaluation Corporation, SPEC CPU2006 Benchmark Suite, 2006. [Online]. Available: https://www.spec.org/cpu2006/

[7] M. W. Ahmed and M. A. Shah, "Cache Memory: An Analysis on Optimization Techniques," *International Journal of Computer and Information Technology*, vol. 4, no. 2, pp. 414, 2015.

[8] L. Eeckhout, "Computer Architecture Performance Evaluation Methods," *Synthesis Lectures on Computer Architecture*, vol. 5, no. 1, pp. 1–146, 2010.

[9] M. Martonosi et al., "Full-system simulation: Techniques, trade-offs, and applications," *IEEE Micro*, vol. 23, no. 5, pp. 34–47, Sept.–Oct. 2003.

[10] S. Eyerman and L. Eeckhout, "Restating the case for quantitative simulations," *IEEE Micro*, vol. 33, no. 4, pp. 31–38, July–Aug. 2013.

[11] D. C. Burger and D. A. Wood, "Accuracy vs. performance in parallel simulation of multiprocessors," in *Proc. 18th Int. Symp. Computer Architecture (ISCA)*, Toronto, ON, Canada, May 1991, pp. 21–30.

[12] G. Einziger and R. Friedman, "TinyLFU: A highly efficient cache admission policy," in *Proc. 22nd Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Xi'an, China, Apr. 2017, pp. 645–658.

[13] M. Kharbutli and Y. Solihin, "Counter-based cache replacement and bypassing algorithms," *IEEE Trans. Computers*, vol. 57, no. 4, pp. 433–447, Apr. 2008.

[14] N. Megiddo and D. S. Modha, "ARC: A self-tuning, low overhead replacement cache," in *Proc. 2nd USENIX Conf. File and Storage Technologies (FAST)*, San Francisco, CA, USA, Mar. 2003, pp. 115–130.

[15] S. Podlipnig and L. Boszormenyi, "A survey of web cache replacement strategies," *ACM Computing Surveys*, vol. 35, no. 4, pp. 374–398, 2003.

[16] G. E. Suh, L. Rudolph, and S. Devadas, "Dynamic Partitioning of Shared Cache Memory," Massachusetts Institute of Technology. [Online]. Available: https://csg.csail.mit.edu/pubs/memos/Memo-522/memo-522.pdf

[17] S. Kumar and P. K. Singh, "An overview of modern cache memory and performance analysis of replacement policies," in *2016 IEEE International Conference on Engineering and Technology (ICETECH)*, pp. 210–215, 2016.

[18] M. Kowarschik and C. Weiß, "An overview of cache optimization techniques and cache-aware numerical algorithms," in *Algorithms for Memory Hierarchies: Advanced Lectures*, U. Meyer, P. Sanders, and J. Sibeyn, Eds. Berlin, Heidelberg: Springer, 2003, pp. 213–232.

[19] D. Lee et al., "Prefetch-aware cache replacement policies for multi-threaded processors," in *Proc. 52nd IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Columbus, OH, USA, Oct. 2019, pp. 201–212.

[20] A. Jain and C. Lin, "Back to the future: Leveraging Belady's algorithm for improved cache replacement," in *Proc. 45th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Vancouver, BC, Canada, Dec. 2012, pp. 78–89.

[21] V. Ahire, S. Patel, and R. Kumar, "Random Adaptive Cache Placement Policy," *arXiv preprint arXiv:2502.02349*, Feb. 2025.

[22] S. Mostofi, S. Gupta, A. Hassani, K. Tibrewala, E. Teran, P. V. Gratz, and D. A. Jiménez, "Light-weight Cache Replacement for Instruction Heavy Workloads," in *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA '25)*, ACM, 2025, pp. 1005–1019, doi:10.1145/3695053.3730993.

[23] S. Eyerman and L. Eeckhout, "System-level performance metrics for multithreaded applications," *IEEE Micro*, vol. 28, no. 3, pp. 42–53, May/June 2008.

[24] E. Hagersten and M. Koster, "WildFire: A scalable path for SMPs," in *Proc. 5th Int. Symp. High-Performance Computer Architecture (HPCA)*, Orlando, FL, USA, Jan. 1999, pp. 172–181.

[25] J. Feliu, A. Perais, D. A. Jiménez, and A. Ros, "Rebasing Microarchitectural Research with Industry Traces," in *2023 IEEE International Symposium on Workload Characterization (IISWC)*, IEEE, Oct. 2023, Ghent, Belgium, doi:10.1109/IISWC59245.2023.00027.

[26] D. H. Albonesi et al., "The Complexity-Effective Processor: A Performance Perspective," in *Proc. 27th Int. Symp. Computer Architecture (ISCA)*, Vancouver, BC, Canada, June 2000, pp. 292–303.

[27] A. Jaleel, "Memory characterization of workloads using instrumentation-driven simulation," *VSSAD Technical Report*, Univ. of Maryland, 2010.

[28] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely, and J. Emer, "SHiP: Signature-based Hit Predictor for high performance caching," in *Proc. 44th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Porto Alegre, Brazil, 2011, pp. 430–441, doi:10.1145/2155620.2155671.

[29] A. Jaleel, K. B. Theobald, S. C. Steely Jr., and J. Emer, "Scavenger: A new last-level cache replacement policy," in *Proc. 37th Annu. Int. Symp. Computer Architecture (ISCA)*, Saint-Malo, France, 2010, pp. 379–390, doi:10.1145/1815961.1816016.

[30] A. Jaleel, K. B. Theobald, S. C. Steely Jr., and J. Emer, "High performance cache replacement using re-reference interval prediction (RRIP)," in *Proc. 37th Annu. Int. Symp. Computer Architecture (ISCA)*, Saint-Malo, France, 2010, pp. 60–71, doi:10.1145/1815961.1815969.

[31] N. Beckmann and D. Sanchez, "LHD: Improving cache hit rate by maximizing hit density," in *Proc. 45th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Vancouver, BC, Canada, 2012, pp. 100–111, doi:10.1145/2382553.2382569.

[32] P. Jain, A. Jain, J. Subramanian, and M. K. Qureshi, "Hawkeye: A learning-based cache replacement policy," in *Proc. 46th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Davis, CA, USA, 2013, pp. 78–90, doi:10.1145/2540708.2540718.

[33] Z. Shi, X. Huang, A. Jain, and C. Lin, "Applying deep learning to the cache replacement problem," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Columbus, OH, USA, 2019, pp. 413–425, doi:10.1145/3352460.3358318.

[34] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely Jr., and J. Emer, "Adaptive insertion policies for high performance caching," in *Proc. 34th Annu. Int. Symp. Computer Architecture (ISCA)*, San Diego, CA, USA, 2007, pp. 381–391, doi:10.1145/1250662.1250709.

[35] M. Peters, N. Gaudin, J. P. Thoma, V. Lapôtre, P. Cotret, G. Gogniat, and T. Güneysu, "On the effect of replacement policies on the security of randomized cache architectures," in *Proc. 19th ACM Asia Conf. Computer and Communications Security (Asia CCS)*, Singapore, 2023, pp. 125–138, doi:10.1145/3579856.3592810.