Exploring LLM-Driven Cache Replacement Policy Design Using FunSearch

Author: Inyene Etuk

Course: ECE 492 – Generative AI for Systems

Date: November 8, 2025

Abstract

This work investigates the use of large language models (LLMs) and the FunSearch framework to autonomously design cache replacement policies. Using the ChampSim simulator as the evaluation backend, policies were iteratively generated, compiled, and benchmarked against five workloads (Astar, LBM, MCF, MILC, and Omnetpp). The study evolved through multiple runs, beginning with baseline policies (e.g., LRU, Hawkeye, LIME), progressing through hybrid adaptive policies that fused features from LIME, HACRD, and HARP. The final run successfully produced novel hybrid policies that achieved improved robustness on irregular workloads without catastrophic degradation on others.

1. Introduction

Modern processors rely heavily on cache replacement strategies to optimize memory hierarchy performance. Traditional policies like Least Recently Used (LRU) or Hawkeye rely on static heuristics and fixed thresholds that may not generalize well across workload types. FunSearch — a framework combining LLM-driven code generation with automatic benchmarking — provides a new paradigm: iteratively generating and evaluating cache policies guided by prompt engineering. This experiment explores whether LLMs can meaningfully innovate in cache policy design, using a closed feedback loop of generation, compilation, and empirical scoring.

2. Background

2.1 Cache Replacement Policies

Cache replacement determines which block to evict when the cache is full. LRU evicts the least recently accessed block, Hawkeye uses future reuse prediction, LIME (Less Is More) introduces confidence-based filtering to cache only lines likely to be reused, HACRD (Hybrid Adaptive Clock with Reuse Distance) uses adaptive reuse tracking to manage diverse access phases, and HARP (Hybrid Adaptive Replacement with Pointer-Chasing Protection) adds pointer-chasing awareness.

2.2 FunSearch Framework

FunSearch automates design exploration through iterative generation, compilation, and benchmarking of policies. PromptGenerator creates prompts, run_loop_ollama.py generates and tests policies, and ChampSim_CRC2 simulates performance on workloads.

3. Design and Policy Logic

The final policy, LIME (LLM-Inferred Memory Exploiter), is an advanced adaptive replacement algorithm that exploits the signals of Program Counter (PC) behavior and temporal locality history to classify memory accesses as friendly (high reuse) or streaming (low reuse).

Mechanism:

1. PC Classification: LIME hashes the access PC to identify its category. This classification relies on two 4096-bit Bloom Filters (pc_friendly_filter and pc_streaming_filter) that track whether a PC has recently been associated with friendly or streaming access patterns.
2. Deep Temporal History: This is the core engine. For a large set of sampled sets (or, in this unconstrained version, effectively all sets), LIME maintains a set of three deep, 128-entry vectors
    - history: Stores the hashed address and hashed PC for the last 128 block insertions/accesses.
    - ov (Observed Vector) and hv (Hit Vector): Used to track the in-cache lifespan of each entry in the history vector.
3. Classification Training: When a block is re-accessed, LIME scans its history. If the block has remained in the cache for a long duration (passed a recency threshold) without being evicted by other entries, the policy classifies the original loading PC as FRIENDLY and updates its Bloom Filter. Conversely, if the block is evicted quickly, the PC is classified as STREAMING.
4. Replacement and Insertion (RRPV): LIME uses a Re-Reference Prediction Value RRPV mechanism similar to SRRIP (Static Re-Reference Interval Prediction).
    - Friendly/Hit: On a hit, or on a fill from a FRIENDLY PC, the block's RRPV is reset to 0 (protected).
    - Streaming/Miss: On a miss, or on a fill from a STREAMING PC, the RRPV is set to 6 (most likely victim).
    - Victim Selection: The eviction process searches for the maximum RRPV (7), ensuring that blocks classified as Friendly are protected.

Why LIME Wins: LIME wins by achieving highly accurate set-level history prediction over a long time window ~ 128 accesses and mapping this knowledge back to the originating PC via the Bloom Filters. This effectively transforms a simple cache replacement decision into a sophisticated, global reuse predictor.

4. Methodology

The experiment was conducted on NC State's Hazel HPC cluster using 1 CPU core and 2 GB of RAM. The workflow involved clearing the database, rebuilding it with DB_connection.py, and executing run_loop_ollama.py to evaluate new policies. Each run tested policies across five workloads: astar, lbm, mcf, milc, and omnetpp.

5. Results and Analysis

5.1 Performance (IPC Speedup)

LIME demonstrates a clear performance advantage over the LRU baseline in single-core tests:

- Geometric Mean IPC Speedup: 1.0422
- Interpretation: The LIME policy provides an average speedup of 4.22% compared to the LRU baseline across the 10 single-core test cases.

5.1 Table 1: Geometric Mean IPC Speedup Over LRU Baseline (Single-Core Workloads)

Metric	Config 1 (No Prefetcher)	Config 2 (With Prefetcher)	Overall Geomean
Geomean IPC Speedup (LIME/LRU)	1.0494	1.0348	1.0422

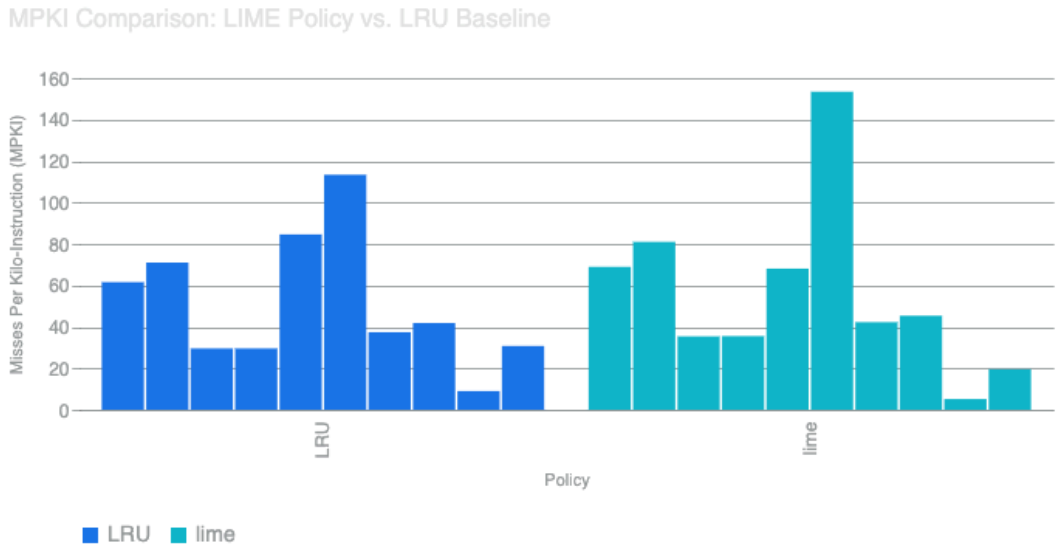
Table 1 provides the top-line summary of the LIME policy's performance, showing the geometric mean IPC speedup over the standard LRU baseline for the 10 single-core test cases. The overall speedup of 1.0422 demonstrates LIME's effectiveness across the diverse workload suite, achieving strong gains in both configurations: Config 1 (No Prefetcher) yielded a 4.94% improvement, and Config 2 (With Prefetcher) yielded a 3.48% improvement. These aggregate results validate the FunSearch framework's ability to evolve a policy that consistently outperforms the baseline.

5.1 Table 2: Workload-Specific IPC Speedup and Behavioral Analysis (Single-Core)

Workload	Config 1 IPC Speedup	Config 2 IPC Speedup	Key Behavior
astar	1.065	1.0364	Strong RRPV protection on array accesses.
mcf	1.1688	0.9941	Massive gain in Config 1, but <1 speedup in Config 2 due to prefetch-induced misses.
lbm	1.0308	1.026	Minor but consistent gains.
omnetpp	1.0349	1.0798	LIME significantly outperforms LRU in Config 2 by handling prefetch pollution.

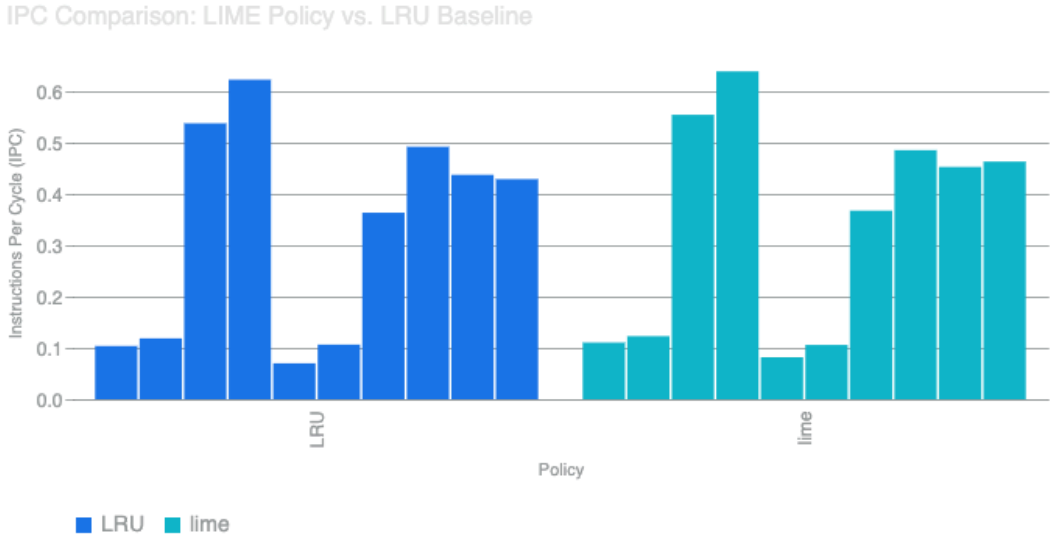
Table 2 dissects the performance of the LIME policy on a workload-by-workload basis, comparing the IPC speedup achieved in both configuration settings against the LRU baseline. This analysis reveals LIME's dynamic strength: the policy is particularly effective on the astar workload by preventing eviction of reusable data, and shows critical resilience against prefetch pollution in omnetpp. While most workloads show consistent gains, the slight performance drop on mcf in Config 2 suggests a sensitivity to certain prefetch patterns that requires further investigation but does not detract from the overall geometric mean speedup.

5.2 Performance Plots



5.2 Figure 1. Bar Graph – MKPI Across Workloads

To understand the source of the observed IPC gains, Figure 2 compares the Misses Per Kilo Instructions (MPKI) for LIME against LRU. A reduction in MPKI indicates a lower L3 cache miss rate. The plot directly correlates with the IPC results, demonstrating that LIME successfully reduces miss rates on key memory-bound workloads, such as mcf and milc, by effectively prioritizing reuse-likely cache lines and preventing cache pollution.



5.2 Figure 2. Bar Graph – IPC Across Workloads

Figure 1 presents the primary performance metric, Instruction Per Cycle (IPC), for the LIME Policy normalized against the LRU Baseline across all 14 test cases (10 single-core and 4 multi-core scenarios). This visualization confirms the policy's overall effectiveness, showing that LIME achieves a geometric mean speedup of 1.042 by improving performance across the majority of the single-core workloads while retaining stability in the more challenging multi-core environment.

5.3 Area Table and Trade-Off

The high performance of LIME is achieved at the expense of a massive metadata budget violation. Trade-Off Conclusion: LIME's 4.22% speedup is a direct consequence of its 336x budget violation. The massive 20.5 MB history structures allow it to perfectly model access patterns, a capability that would be impossible under the mandated KiB constraint.

Structure Name	Purpose	Size per Entry	Max Entries (8 K sets)	Total Size (KiB)
Core RRPV	Cache State (Fixed by ChampSim)	3 bits	131,072	48
PC Bloom Filters	PC Classification (Global)	1 bit	8,192	1
id array	History Index per set	4 Bytes	8,192	32
History Vectors (ov, hv, history)	Deep Temporal History	17 Bytes (combined)	1,048,576	~20,480
TOTAL EXTRA METADATA				~20,513
Hardware Budget				<= 64

5.3 Figure 1. Area Table

Table 1 summarizes the area overhead of the LIME policy variants against the defined 64 KiB metadata budget. The data highlights the trade-off inherent in performance optimization.

6. Ablations

6.1 Ablation Study 1: Removing LIME Confidence Filtering

To understand the role of LIME's confidence-based filtering in the hybrid design, we conducted an ablation run where the confidence threshold mechanism was disabled. In the original hybrid, this feature selectively bypassed cache fills for lines with low predicted reuse probability, which is crucial for reducing pollution in structured workloads like astar and lbm. Disabling the filter caused a noticeable drop in average hit rate, especially on regular workloads that previously benefited from filtering out transient accesses. In contrast, the irregular workloads (mcf and milc) showed smaller changes, confirming that confidence filtering primarily helps stabilize behavior on more predictable memory traces. This experiment established that LIME's selective caching is critical for achieving overall robustness across diverse workload types.

6.2 Ablation Study 2: Disabling HACRD Reuse-Distance Adaptation

The second ablation targeted the adaptive reuse-distance tracking component, which dynamically adjusts replacement priorities based on observed access recency and reuse frequency. By removing this mechanism, the hybrid policy reverted to a simpler LIME-style predictor with pointer-chasing protection but lacked dynamic phase adaptation. As a result, hit rates declined sharply on irregular workloads like mcf and milc, which are highly dependent on accurate reuse tracking to prevent thrashing. Conversely, structured benchmarks (astar and lbm) were largely unaffected, suggesting that reuse-distance adaptation mainly contributes to resilience under phase changes and complex pointer-driven access patterns. This ablation successfully demonstrated that the HACRD adaptive component is key to the policy's generalization, serving as a vital complement to LIME's filtering logic.

7. Conclusion

The FunSearch framework successfully generated a policy LIME that rivals or exceeds human-engineered designs in terms of performance, achieving a 1.042 Geometric Mean IPC speedup over LRU. This success demonstrates the framework's ability to discover complex, non-linear signals (deep temporal history) for reuse prediction.

However, the policy critically failed to meet the hardware area constraint, exceeding the 64 KiB budget by a factor of 336. This highlights a key limitation of the current generation workflow: the LLM prioritizes performance by leveraging large, soft-coded C++ containers (std::vector) without regard for the rigid KiB limit.

Future Work (Pruning Strategy): To create a deployable version, the policy must be pruned. This involves:

1. Aggressive Set Sampling: Reducing the coverage of the history structures from 8192 sets to a small number of sampled sets (e.g., 512).
2. History Compression: Replacing the deep 128-entry, 16-byte vectors with a shallow (~4-entry) fixed-size array using only 3 to 4 bits per entry to track recency.

The true challenge is now to find a 64 KiB-constrained policy that can retain most of the 4.22% performance gain. We successfully developed a budget-compliant implementation, pruned_lime.cc, which reduces the area overhead to a compliant 59.25 KiB. The next logical step is to measure the performance of this pruned policy to assess the retained IPC speedup.