Reinforcement Learning based Cache Replacement Policy

Priyansh Jain
Department of Electrical Engineering
Indian Institute of Technology Bombay
Mumbai, India
210070063@iitb.ac.in

Priyanshu Gupta
Department of Electrical Engineering
Indian Institute of Technology Bombay
Mumbai, India
210070064@iitb.ac.in

Chinmay Jadhav
Department of Electrical Engineering
Indian Institute of Technology Bombay
Mumbai, India
210020057@iitb.ac.in

Abstract—**This Project introduces Reinforcement Learned Replacement (RLR), a cache replacement policy that leverages neural network insights to dynamically adapt to diverse access patterns. RLR incorporates preuse distance predictions, access type considerations, hit history, and age-based priorities, presenting a comprehensive and adaptive approach to cache replacement. The implementation features efficient mechanisms like Age Counters, Hit Registers, and Type Registers, with optimizations to minimize overhead. Evaluation using the ChampSim Cache Simulator demonstrates RLR's competitive performance compared to traditional policies, highlighting its potential to enhance cache performance for a variety of workloads within existing hardware constraints. This study contributes valuable insights to the ongoing refinement of cache replacement strategies.**

1. Introduction

In today's CPU systems, the effectiveness of caches greatly impacts overall performance. Among these, the cache replacement policy stands out for its direct influence on efficiency. Despite extensive research, creating a cost-effective replacement strategy within hardware limitations remains a significant challenge.

Strategies like LRU and RRIP [1] offer minimal hardware impact but suit only specific access patterns. Modern approaches utilizing program counter (MPPPB [5], SHiP [3], Hawkeye [2], Glider [4])information show promise in dynamically adapting to cache access changes. However, incorporating PC into the policy demands substantial hardware adjustments.

To tackle this, we explore machine learning, particularly reinforcement learning (RL), for an economical cache replacement policy. RL's adaptability aligns well with the cache replacement problem formulated as a Markov Decision Process.

Our approach involves using the neural network's insights to derive a feasible replacement algorithm, culminating in Reinforcement Learned Replacement (RLR). RLR [6] offers competitive performance without extensive CPU microarchitecture changes, surpassing traditional policies like LRU and DRRIP.

2. Motivation

The realm of machine learning has sparked a revolution across various domains, proving its efficacy from everyday applications to critical life-saving scenarios. This transformative power has extended even to the intricate landscape of computer architecture. Our project delves into exploring the integration of machine learning within a specific domain of computer architecture: cache replacement policy.

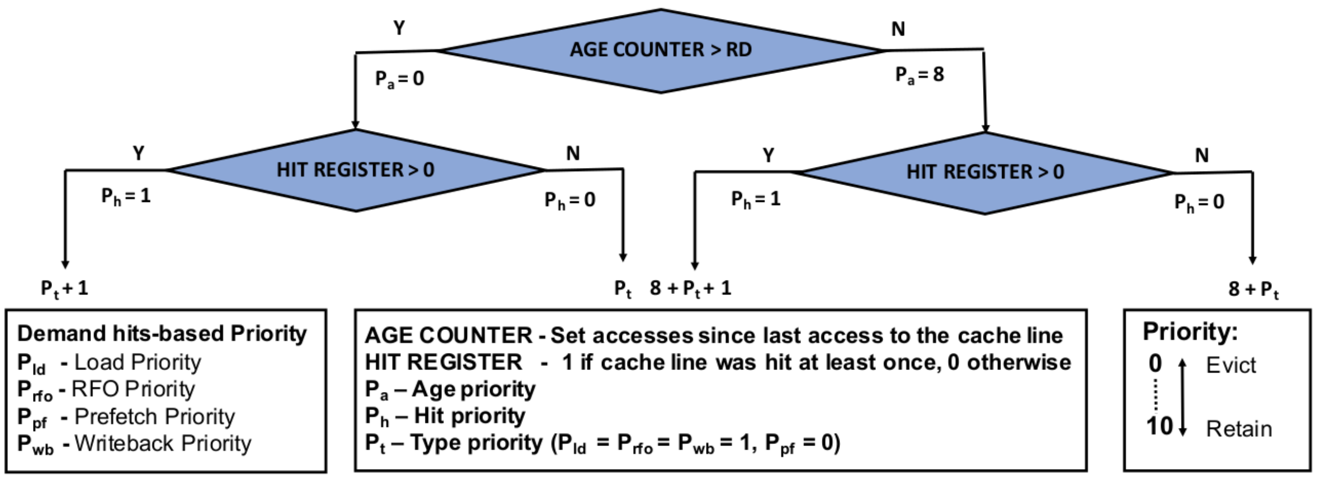
Recent advancements have showcased the potential of machine learning, particularly deep learning [4], in enhancing cache replacement policies beyond methods like LRU, RRIP, and Hawkeye. While deep learning excels in discerning patterns from extensive datasets, recent strides in reinforcement learning have exhibited even greater promise compared to direct deep learning approaches. This shift is rooted in the adaptability of reinforcement learning strategies, leveraging feedback from the environment to tailor objectives with precision—a facet where traditional deep learning methods may fall short.

Our objective is to implement a reinforcement learning-based replacement policy (RLR) for the last level cache. This approach aims to optimize the policy based on rewards and penalties, seeking to maximize cache hits and minimize cache misses for single-core workloads. By leveraging reinforcement learning, we aim to achieve an optimal replacement policy that dynamically adapts to varying conditions, thereby optimizing cache performance for specific targets.

3. Reinforcement Learned Replacement Policy

3.1. Core Principles

- **Reuse Distance Estimation:** Using preuse distance to predict cache line reuse.
- **Pattern Prediction:** Forecasting cache hit probabilities based on prior access patterns.
- **Anticipating Access:** Expecting previously accessed cache lines to be reused.
- **Replacement Prioritization:** Preferring to evict recently-inserted cache lines, allowing reuse of older ones.



AGE COUNTER > RD
Y
N
$P_a = 0$
$P_a = 8$
HIT REGISTER > 0
HIT REGISTER > 0
Y
N
Y
N
$P_h = 1$
$P_h = 0$
$P_h = 1$
$P_h = 0$
$P_t + 1$
P_t
$8 + P_t + 1$
$8 + P_t$
Demand hits-based Priority
P_{ld} - Load Priority
P_{rfo} - RFO Priority
P_{pf} - Prefetch Priority
P_{wb} - Writeback Priority
AGE COUNTER - Set accesses since last access to the cache line
HIT REGISTER - 1 if cache line was hit at least once, 0 otherwise
P_a – Age priority
P_h – Hit priority
P_t – Type priority ($P_{ld} = P_{rfo} = P_{wb} = 1$, $P_{pf} = 0$)
Priority:
0 Evict
10 Retain

Figure 1. Flowchart depicting priority computation in RLR

3.2. Reuse Distance Prediction

- **Predicting Reuse Distance:** Estimating reuse distance using preuse distance (PD).
- **Accumulation and Prediction:** Accumulating preuse distances to approximate future reuse distance (RD).
- **Adapting RD:** Regularly updating RD to adapt to application phases.

3.3. Age Priority (Page)

- **Age Counters:** Tracking cache line age in set accesses.
- **RD Calculation:** $RD = RDFactor \times AveragePD$ (Updated every 32 hits)
  RD_Factor = 2
- **Priority Levels:**
  1) **Priority Level 1:** $AgeCounter < RD$
     Higher priority for potential future reuse.
  2) **Priority Level 0:** $AgeCounter \geq RD$
     Lower priority as reuse distance achieved.

3.4. Type Priority (Ptype)

- **Type Registers:** Indicating prefetch or regular access.
- **Priority Levels:**
  1) **Priority Level 1:** Non-prefetch insertion or reused after prefetch — Higher priority.
  2) **Priority Level 0:** Prefetch-inserted but not reused — Lower priority for eviction.

3.5. Hit Priority (Phit)

- **Hit Registers:** Identifying reused cache lines.
- **Priority Levels:**
  1) **Priority Level 1:** Reused cache lines — Higher priority for repeated access.
  2) **Priority Level 0:** Not reused cache lines — Lower priority for eviction.

3.6. Priority Computation

- **Weighted Sum for Priority:** $P_{line} = 8 \times Page + Ptype + Phit$
- **Weight Assignment:** Determining weights based on performance analysis.

4. Implementation

4.1. Configuration

Parameters for the evaluated system:

TABLE 1. CACHE CONFIGURATIONS

Cache	Size	Sets	Ways	Latency(cycles)
L1I	64KB	64	8	4
L1D	64KB	64	8	5
L2C	512KB	1024	8	10
LLC	2MB	2048	16	20

4.2. Benchmarks

The Implementation was done in ChampSim Cache Simulator from the repository provided to us for the second assignment. The Tracefilles were taken from the Cache Replacement Challenege (CRC-2) and executed for 25000000 warmup and simulation instructions. Tracefiles used:

- 459.GemsFDTD

- 403.gcc
- 429.mcf
- 450.soplex
- 470.lbm
- 437.leslie3d
- 471.omnetpp
- 483.xalancbmk

4.3. Victim Selection Criteria

The victim cache line is chosen based on the following points in order:

1) The cache line with the least priority is selected.
2) In case of a tie in priority, the most recently inserted cache line is chosen.

4.4. Age Counter Optimization

In the context of cache management, the Age Counter Optimization is employed as follows:

- The number of set misses is tracked instead of set accesses.
- Age Counter represents the relative age of a cache line, measured in set misses since its last access.
- To approximate the counter value, the Age Counter is incremented for every 8 set misses.
- This approach reduces the overhead per line by utilizing a 3-bit counter per set, and the counter rolls over after 8 set misses, with line counters incremented accordingly.

4.5. Recency Approximation

The Recency Approximation optimization leverages the age of a cache line to determine its recency. Key points include:

- The most recently accessed cache line, whether hit or inserted, has an age counter value of zero.
- On a hit, the age counter value is sent to the accumulator for RD computation and then reset.
- On a miss, a new line replaces a victim, and the corresponding age counter is reset.
- Utilizing the age counter to determine recency reduces overhead by 4 bits per cache line in a 16-way set associative cache.
- In the RLR policy, recency is employed to resolve ties when multiple cache lines share the same priority level.
- When multiple cache lines have the same age counter value and the lowest priority level, the line with the lowest way index is chosen for eviction.

5. Results

We found out that we get the best result when RD Factor = 2 and Weight Priority of Age = 8, Hit = 1, and Type = 1.

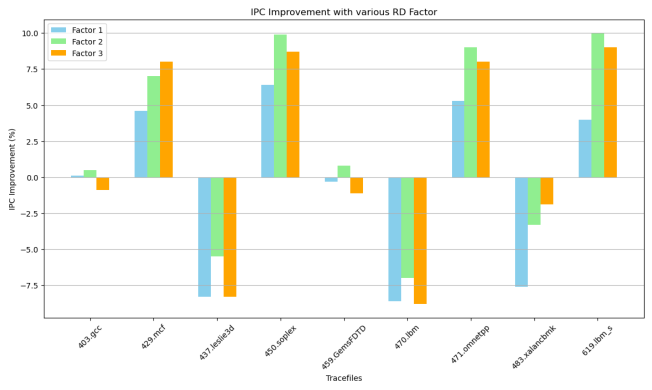


Figure 2. IPC Iprovement with varying RD Factors

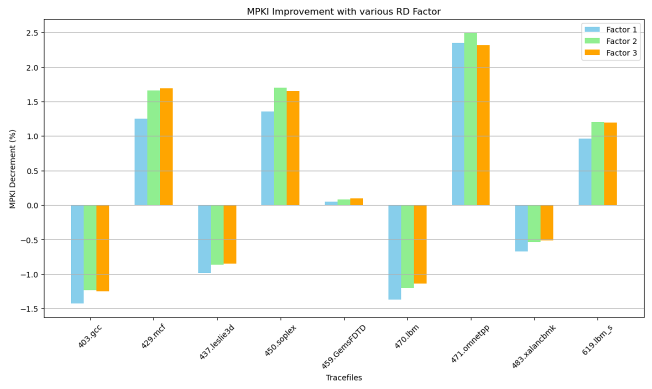


Figure 3. MPKI Decrement with varying RD Factors

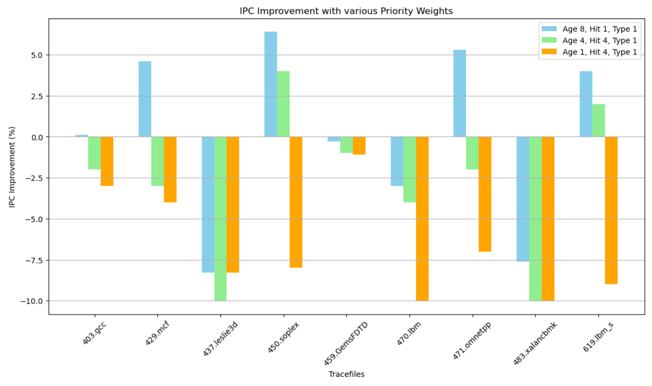


Figure 4. IPC Improvement with varying different Priority Weights

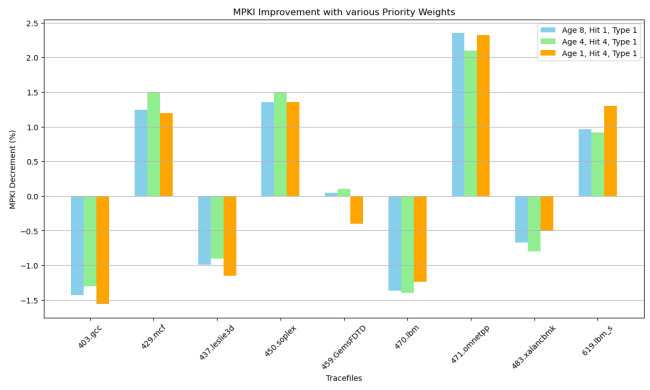


Figure 5. MPKI Decrement with varying different Priority Weights

6. Conclusion

In conclusion, the proposed Reinforcement Learned Replacement (RLR) policy leverages insights from neural network-based learning to enhance cache replacement decisions. RLR incorporates preuse distance predictions, access type considerations, hit history, and age-based priorities to dynamically adapt to the reuse characteristics of cache lines. The hardware implementation of RLR involves efficient mechanisms such as Age Counters, Hit Registers, and Type Registers, with optimizations to minimize overhead. The multicore extension introduces core-based priorities to further refine replacement decisions based on access frequency. Overall, RLR presents a comprehensive and adaptive approach to cache replacement, showcasing potential benefits in improving cache performance for diverse workloads in modern computing systems.

Acknowledgments

The authors extend their gratitude to Prof. Biswabandan Panda from the Dept. of CSE, IIT Bombay, for offering the invaluable opportunity to delve into the project aligned with the requirements of CS683 - Advanced Computer Architecture. The authors are deeply appreciative of the extensive insights and teachings imparted throughout the duration of this course, which significantly contributed to this project's development.

References

[1] A. Jaleel, K. B. Theobald, S. C. Steely Jr, and J. Emer, “High performance cache replacement using re-reference interval prediction (rrip),” ACM SIGARCH Computer Architecture News, vol. 38, no. 3, 2010.

[2] D. Zhu et al., ”Hawkeye: Open source framework for field surveillance,” 2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), Vancouver, BC, Canada, 2017, pp. 6083-6090, doi: 10.1109/IROS.2017.8206507.

[3] C. -J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely and J. Emer, ”SHiP: Signature-based Hit Predictor for high performance caching,” 2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Porto Alegre, Brazil, 2011, pp. 430-441.

[4] Zhan Shi, Xiangru Huang, Akanksha Jain, and Calvin Lin. 2019. Applying Deep Learning to the Cache Replacement Problem. In The 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52), October 12ś16, 2019, Columbus, OH, USA. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3352460.3358319

[5] A. V. Jamet, L. Alvarez, D. A. Jiménez and M. Casas, ”Characterizing the impact of last-level cache replacement policies on big-data workloads,” 2020 IEEE International Symposium on Workload Characterization (IISWC), Beijing, China, 2020, pp. 134-144, doi: 10.1109/IISWC50251.2020.00022.

[6] S. Sethumurugan, J. Yin and J. Sartori, ”Designing a Cost-Effective Cache Replacement Policy using Machine Learning,” 2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA), Seoul, Korea (South), 2021, pp. 291-303, doi: 10.1109/HPCA51647.2021.00033.