Optimization of Hardware Cache Management Using AI

Ajayanath Chittela, Keerthana Mohana, Suraj Patel Muthe Gowda, Naveen Naik Sapavath
Dept of Electrical and Computer Engineering
Northeastern University
Boston, USA
Email: {ajayanath.c, mohana.k, muthegowda.s, n.sapavath}@northeastern.edu

***Abstract*—Cache replacement policies play a critical role in determining the performance of modern computing systems, as they directly impact the efficiency of memory access and overall system throughput. Traditional cache replacement policies, such as Least Recently Used (LRU) and First-In-First-Out (FIFO), often fall short in adapting to dynamic and complex workload patterns, leading to suboptimal performance. To address these limitations, adaptive AI/ML-based algorithms have emerged as promising alternatives, leveraging data-driven insights to make intelligent replacement decisions. Among these, reinforcement learning (RL)-based algorithms have shown significant potential, delivering better results compared to traditional policies by dynamically adapting to workload characteristics. This study explores the design and evaluation of an RL-based cache replacement algorithm, demonstrating its effectiveness in improving system performance metrics such as hit/miss rates and memory latency.**

***Index Terms*—Cache, Cache replacement policies,**

I. Introduction

Modern computer systems must have effective memory management because overall performance is greatly impacted by the growing disparity between processor speeds and memory access times. By keeping frequently accessed data closer to the processor, cache memory acts as a crucial middleman, lowering latency and improving system performance. However, a cache's replacement strategy, which chooses which data to remove when the cache fills up, has the biggest impact on how effective the cache is. Traditional replacement policies use predefined algorithms that are adapted to certain workload factors. Examples of these policies are Least Recently Used (LRU) and First-In First-Out (FIFO). These policies function well when access patterns are predictable, but because they are static, they cannot adjust to workloads that are dynamic or change quickly. This limitation often results in suboptimal performance, more cache misses, and reduced system efficiency.

To overcome these restrictions, this study presents a novel adaptive cache system that leverages reinforcement learning (RL) approaches to dynamically adjust its replacement policy in real time. At its core, the system intelligently learns from performance metrics, such as cache hit and miss rates, as well as workload characteristics, to optimize its policy decisions. Unlike traditional approaches, which rely on static predefined algorithms, this RL-based framework adapts to varying workload behavior and improves cache efficiency by continuously optimizing data retention and eviction strategies.

The design also allows for user-defined characteristics, such as cache size, offering the flexibility to tailor the cache system to a wide range of operating systems and applications. This adaptability ensures that the proposed solution is suitable for diverse scenarios, from high-performance computer systems with complex memory requirements to embedded systems with stringent latency demands.

By integrating reinforcement learning into cache management, the proposed adaptive cache system transcends the inherent rigidity of static replacement approaches. It offers a scalable, workload-aware solution that effectively manages the challenges posed by unpredictable or dynamic workloads. With its real-time learning and adaptation capabilities, the system consistently delivers superior performance across a wide variety of computational environments, marking a significant advancement in modern memory management strategies.

II. Experimentation and Methodology

This section describes the structured approach used to optimize hardware cache management. The methodology encompasses multiple phases, starting from trace file generation to evaluating traditional and AI-enhanced replacement policies, with a specific focus on employing reinforcement learning for dynamic cache optimization. By analyzing diverse workload patterns, the study seeks to identify the limitations of existing strategies and propose innovative solutions. The ultimate goal is to enhance cache efficiency while maintaining adaptability to varied and unpredictable memory access behaviors.

A. *Trace File Generation*

Trace file generation serves as the foundational step in this experiment, creating a detailed record of memory access patterns. These trace files are indispensable for simulating cache behavior, analyzing memory management strategies, and evaluating the performance of replacement policies.

1) Program Development: In order to capture different memory access patterns, several C programs were written to simulate realistic computational scenarios. Each program was designed to highlight certain types of memory operations and access patterns, including:

Identify applicable funding agency here. If none, delete this.

- Arithmetic Operations: Programs performing basic computations to evaluate sequential and predictable memory accesses.
- Conditional Logic: C programs with conditional statements (if-else and switch) to introduce branch-based access variations.
- Matrix Multiplication: A program involving nested loops for linear access patterns across large data blocks.
- Memory Allocation: Programs that dynamically allocate and deallocate memory using `malloc` and `free` to simulate memory-intensive operations.
- Nested Loops: Examples with multiple layers of loops to observe compounded memory accesses.
- Pointer Dereferencing: Programs accessing memory locations indirectly through pointers to mimic complex memory referencing.
- Random Array Access: Simulations of irregular access patterns using random index selections in arrays.
- Struct Object Access: Programs manipulating user-defined data structures to evaluate their memory access behavior.
- System Calls: Programs incorporating system-level operations for evaluating instruction fetches and memory utilization.
- Variable Loads: Code manipulating a high volume of variables in different execution contexts.

These programs were written in C, compiled, and executed to produce memory traces that represent their runtime behavior.

2) Trace Extraction Using Valgrind-Lackey Tool: The execution of these programs was analyzed using Valgrind, a memory profiling tool that is part of the Valgrind suite. Valgrind records detailed information about memory operations, including reads, writes, and instruction fetches. The key features of the Valgrind tool include:

- Memory Access Profiling: Tracks every memory access performed by the CPU, providing insights into load and store operations.
- Instruction Fetching: Captures the program counter's activity to document how instructions are fetched from memory.
- Granular Data Representation: Outputs memory operation details in a structured format for detailed analysis.

When the programs were executed through Valgrind, it generated traces with entries as observed in Fig. 1.

Fig. 1 shows a sample of different traces generated using the Valgrind tool. A value of 0 represents a data read, 1 represents a data write, 2 represents an instruction fetch, and the hexadecimal numbers indicate memory addresses where these operations occurred.

3) Formatting the Output: The trace files produced by Valgrind were formatted to suit the input requirements of cache simulation tools. Each trace captured:

- The type of operation (load, store, or fetch).
- The memory address accessed.
- The sequence of operations.

```
2 04001143
1 fff000278
2 04001850
1 fff000270
2 04001851
2 04001854
1 fff000268
2 04001856
1 fff000260
2 04001858
```

Fig. 1. Generated Trace Snippet

This consistent formatting allowed seamless integration with Valgrind, the simulation environment used later in the study.

B. Cache Simulation Using Python Scripts

Simulating cache behavior was crucial to understanding the effects of configuration parameters and replacement policies on system performance. Python-based simulation scripts were used to implement and evaluate traditional cache replacement strategies such as Least Recently Used (LRU) and First-In-First-Out (FIFO). These simulations used a fixed 256-byte cache size, mimicking the constraints found in embedded systems, and were evaluated on trace files extracted from real-world programs. The simulation setup included the following steps:

1) Cache Size and Configuration:

- A 256-byte cache size was chosen, representing a constrained environment typical of embedded systems.
- The experiments focused on comparing the efficiency of various replacement policies under the same cache size, ensuring consistent results.

2) Trace Files:

- The trace files generated during the earlier phase of experimentation served as the input.
- 0 for instruction read, 1 for instruction write, and 2 for instruction fetch.
- Each memory address (e.g., fff000278, 04001850) was processed sequentially to simulate memory accesses.

*3) **Replacement Policies Simulation**:*

- **Least Recently Used (LRU)**: Evicts the block that has not been accessed for the longest time, leveraging temporal locality. The LRU policy was implemented using a Python class, `LRUCache`, which employed an ordered dictionary (`OrderedDict`) to maintain the access order of elements. The working mechanism includes:

$$\text{LRU\_Eviction} = \arg\min_{i}(\text{Recency}(Key_i))$$

  Where:
  - Recency(Key_i) represents the access timestamp for the cache entry associated with Key_i.

- $\arg\min_i$ selects the key with the oldest access time to be evicted, which is the least recently used.
- Cache Access - If the memory address (key) exists in the cache, it is a cache hit. The key is moved to the end of the dictionary to mark it as recently accessed. If the key is not found, it is a cache miss. The key-value pair is added to the cache, and if the cache exceeds its capacity, the least recently used item (the first in the dictionary) is evicted.
- Eviction - When the cache reaches its capacity, the oldest accessed key-value pair is removed.

The strengths of this policy are:

- Leverages Temporal Locality: Ensures frequently accessed memory blocks remain in the cache.
- Efficient for Mixed Workloads: Particularly effective when memory access patterns shift between sequential and random.

The disadvantages of LRU mainly include:

- Overhead of Tracking Access Order: Managing access order in hardware can be complex and resource-intensive.
- Potential Suboptimal Performance for Cyclic Patterns: Inefficient for cyclic memory patterns where blocks are reused after a long interval.

- **First-In-First-Out (FIFO)**: Evicts the oldest block in the cache, based on the order of arrival. The FIFO policy was implemented using a Python class, `FIFOCache`, which used a `deque` to maintain the order of arrival of elements. The working mechanism includes:
  - Cache Access - If the memory address (key) exists in the cache, it is a cache hit. If not, it is a cache miss. The key-value pair is added to the cache, and if the cache exceeds its capacity, the first item in the queue is evicted.
  - Eviction - The oldest key-value pair, based on insertion order, is removed when the cache size exceeds its capacity.

  The strengths of the FIFO policy mainly include:
  - Simplicity: Easy to implement and maintain due to its straightforward eviction rule.
  - Effective for Sequential Workloads: Performs well for memory patterns with predictable sequential access.

  The FIFO policy is disadvantageous in terms of:
  - No Temporal Locality: Ignores the recency of access, leading to frequent evictions of frequently used blocks.
  - Suboptimal for Random or Mixed Workloads: Poor adaptability to irregular or hybrid memory access patterns.

By simulating these policies, the limitations of traditional approaches were observed, paving the way for implementing advanced AI-based strategies like reinforcement learning for dynamic cache optimization.

4) Analysis, AI Integration, and Reinforcement Learning: This phase marked a significant leap in the optimization of cache management by analyzing traditional policies, designing an adaptive strategy, and integrating artificial intelligence (AI) models like Reinforcement Learning (RL) and Perceptron-based Replacement Policies. These innovative approaches aimed to dynamically adapt to diverse workloads and overcome the limitations of conventional strategies.

a) Analysis of Traditional Strategies:: The analysis of traditional cache replacement strategies, namely LRU (Least Recently Used) and FIFO (First-In-First-Out), revealed that no single strategy could consistently deliver optimal performance across a wide range of workloads. Each of these policies exhibited strengths in certain scenarios, but also had significant limitations when faced with workloads that did not align with their inherent assumptions.

- **Least Recently Used (LRU):**
  - **Strengths**
    - High Temporal Locality Handling: LRU excels in environments where temporal locality is prominent—meaning that once a memory block is accessed, it is likely to be accessed again in the near future. LRU maintains a list of blocks in the cache, with the most recently accessed blocks at the ”end” of the list and the least recently accessed blocks at the ”front.” This strategy effectively evicts the least recently used block, making it ideal for applications where frequently used data tends to remain in the cache.
    - Performance in Predictable Workloads: For workloads that demonstrate strong temporal locality, such as matrix multiplications or recursive algorithms, LRU consistently achieves high cache hit rates, as the blocks that were most recently used are retained for future accesses.

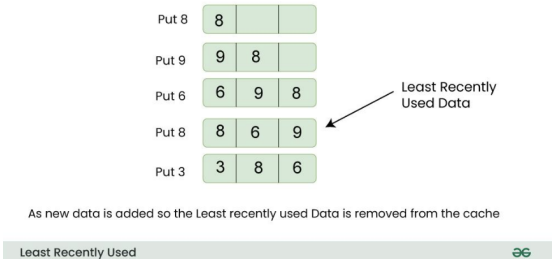

Fig. 2. LRU Replacement Policy

  - **Weaknesses**
    - Poor Performance in Random or Cyclic Access: LRU’s performance significantly degrades in environments where memory accesses are random or cyclic. In these workloads, memory blocks that were recently used are frequently evicted, even though they may be needed again shortly. This is particularly problematic for applications involving

graph traversals, hash tables, or other irregular access patterns. Here, LRU may evict a memory block that will soon be accessed again, resulting in unnecessary cache misses and increased access time.

* High Overhead in Maintenance: For certain workloads, LRU can incur significant overhead in tracking the access history of each cache block. This is because LRU must continuously reorder the cache to maintain access order, which can be computationally expensive, particularly when the cache size is large or the access pattern is complex.

- **First-In-First-Out (FIFO):**
  - **Strengths**
    * Simplicity and Efficiency for Sequential Access: FIFO is one of the simplest cache replacement policies, operating on the basic principle of evicting the oldest cache entry in a straightforward queue-like fashion. Its implementation is minimal and computationally inexpensive. FIFO is particularly effective in sequential access patterns, where memory blocks are accessed in a predictable, linear order. In such cases, FIFO ensures that the oldest blocks are evicted in favor of newer ones, which aligns well with the workload's behavior.
    * Predictability in Sequential Workloads: For tasks such as file processing or data streaming, where each memory block is used once and then discarded, FIFO maintains low miss rates and reduces the risk of unnecessary evictions. By evicting the oldest blocks first, FIFO can effectively handle the sequential nature of such workloads without causing excessive misses or access penalties.

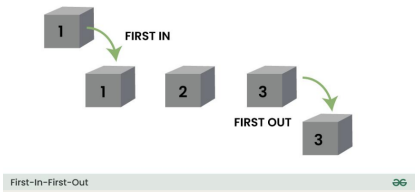

Fig. 3. FIFO Replacement Policy

  - **Weaknesses**
    * Inability to Adapt to Mixed or Random Access Patterns: FIFO's greatest limitation is its inability to adapt to workloads that exhibit mixed or random access patterns. Unlike LRU, which takes recent access history into account, FIFO relies solely on the order in which memory blocks were added to the cache. As a result, it is unable to prioritize blocks that are more likely to be accessed again, leading to inefficient use of cache space. This results in an increase in cache misses in workloads that combine sequential, random, and irregular access patterns, such as dynamic memory allocation, pointer dereferencing, or non-linear data structures.
    * Fixed Eviction Criteria: FIFO operates on a rigid eviction policy, where the oldest block is evicted regardless of how frequently or recently it was accessed. This lack of adaptability means that in certain workloads, valuable memory blocks that have been accessed multiple times might be evicted simply because they were inserted earlier. In contrast, a policy like LRU could have retained these frequently used blocks, improving the cache hit rate.

- **Adaptive Cache Replacement Policy: A Dynamic Approach** - The Adaptive Cache Replacement Policy introduces a dynamic mechanism to mitigate the limitations of static replacement policies like Least Recently Used (LRU) and First-In-First-Out (FIFO). This policy intelligently switches between the two strategies based on real-time performance metrics, optimizing cache management for varying workloads. By dynamically adjusting its mode of operation, the adaptive policy aims to balance hit rates and response times across diverse memory access patterns.
  - **Design and Key Features** - The adaptive policy alternates between LRU and FIFO modes based on observed cache miss behavior. This switching is governed by a miss threshold, a pre-defined parameter that triggers a mode change after a certain number of consecutive cache misses. The implementation followed the equation:

$$\text{Mode}_{\text{adaptive}} = \begin{cases} \text{LRU}, & \text{if Miss\_Count} \leq \text{Threshold} \\ \text{FIFO}, & \text{if Miss\_Count} > \text{Threshold} \end{cases}$$

$$\text{Miss\_Count} = \sum_{i=1}^{N} \mathbb{I}(Access[i] = \text{Miss})$$

Where $\mathbb{I}(\cdot)$ is an indicator function that returns 1 if the condition is true (cache miss), otherwise 0. The cache mode switches when the miss count exceeds the threshold, adapting to the access pattern for better performance.

  - **Modes of Operation:**
    * **LRU Mode** - Tracks the recency of accesses and prioritizes retaining recently used blocks in the cache. Suitable for workloads with strong temporal locality.
    * **FIFO Mode** - Maintains a strict insertion order, evicting the oldest blocks first. Effective for sequential workloads with limited temporal locality.
  - **Dynamic Switching:**

    - The policy continuously monitors the miss rate during cache operations.
    - If the number of consecutive misses exceeds the threshold, the policy switches modes, assuming the current strategy is suboptimal for the workload.
    - Switching involves converting the active cache state to the new mode, ensuring no disruption in data retention.
  - **Performance Metrics:**
    - Tracks total cache accesses, hits, and misses.
    - Calculates miss rates to evaluate mode performance and trigger mode changes.
    - Provides insights into the workload patterns by analyzing cache behavior over time.
  - **Adaptability:**
    - LRU is favored for workloads dominated by repeated access to a small working set, while FIFO is preferred for workloads with sequential data access.
    - By switching dynamically, the adaptive policy avoids prolonged inefficiencies caused by sticking to a single static strategy.
  - **Limitations and Insights** - While the adaptive policy proved effective in handling most workloads, certain limitations were observed:
    - **Overhead**: The mode-switching mechanism introduced minor computational overhead, which could marginally impact performance in highly constrained environments.
    - **Miss Threshold Sensitivity**: The selection of an appropriate miss threshold was crucial. An overly sensitive threshold led to frequent, unnecessary switches, while an insensitive one delayed beneficial mode changes.
    - **Dynamic Workloads**: Although the adaptive policy was versatile, workloads with highly dynamic and unpredictable patterns still posed challenges, suggesting the need for further optimization.

  The adaptive policy represents a significant step toward intelligent cache management by leveraging runtime observations to adjust to workload demands. However, its limitations indicate potential for further enhancement.
- **Perceptron Based Cache Replacement Strategy** - The Perceptron algorithm is a simple yet powerful machine learning model that has been used in many binary classification tasks. This algorithm serves as a linear classifier mapping input features to binary outputs. In adaptive cache replacement strategies, the perceptron algorithm has been used to make intelligent eviction decisions based on multiple features of the cache accesses. The key goal was the use of the perceptron to predict whether a cache block should be replaced or not, based on the history of cache access patterns, which includes frequency, recency, and presence of blocks in the cache.

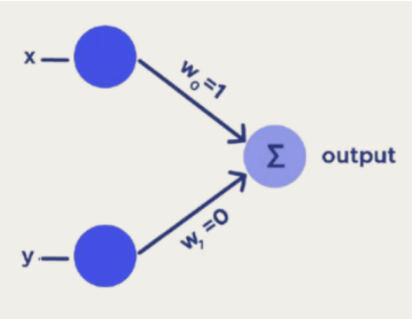

Fig. 4. Perceptron

  - **Model Overview**: The perceptron is a type of single-layer neural network that makes decisions based on weighted combinations of input features. In the case of adaptive cache replacement, the perceptron predicts the likelihood of eviction for a given cache block based on several input features derived from the current state of the cache and the block being accessed. The perceptron model uses three key features to make eviction decisions:
    - **Frequency** - The number of times a particular cache block has been accessed. Frequently accessed blocks are less likely to be evicted.
    - **Recency** - The position of the block in the access history queue, indicating how recently it was accessed. Blocks that have been accessed more recently are less likely to be evicted.
    - **Presence** - A binary feature indicating whether the block is present in the cache at the time of access.

    These features are passed through a single-layer perceptron, which applies a linear transformation followed by a sigmoid activation function to produce a value between 0 and 1, indicating the likelihood of eviction. A higher value signifies that the block is more likely to be evicted. The perceptron outputs a probability (between 0 and 1) of whether the cache block should be evicted when the cache is full. A threshold is used to convert this probability into a binary decision: if the probability is greater than a set threshold, eviction is triggered.
  - **Adaptive Cache with Perceptron Integration**: The adaptive cache system integrates the perceptron model to dynamically adjust cache management based on ongoing memory access patterns. The system tracks access history and utilizes the perceptron to predict eviction decisions.
    - **Accessing the Cache**: When a new memory address is accessed, the system first checks whether the address is already in the cache. If the address is present, it is considered a cache hit, and its frequency and recency are updated accordingly. If the address is not in the cache, it is a cache miss,

and the system then decides whether to evict an existing block.

* **Eviction Prediction**: When the cache is full, the perceptron model is used to decide which block to evict. The features for each block (i.e., frequency, recency, and presence) are fed into the perceptron, which computes a probability for each block. If the eviction probability exceeds a pre-defined threshold, the system evicts the block predicted by the perceptron. The model continuously learns from cache accesses and eviction decisions, refining its predictions over time.

The basic formula for a perceptron is:

$$y = f\left(\sum_{i=1}^{n} w_i x_i + b\right)$$

where the activation function $f(z)$ is defined as:

$$f(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases}$$

- **Training the Perceptron**: The perceptron model is trained using the Binary Cross-Entropy Loss function, a common loss function for binary classification problems. The training process involves adjusting the weights of the perceptron based on the features of cache accesses and the labels indicating whether an eviction was necessary.
  * **Training Data**: The training data consists of recent cache access events, including the features of each access (frequency, recency, and presence) and the label indicating whether eviction occurred (1 for eviction needed, 0 for no eviction). As cache accesses continue, the system gathers more data to improve the perceptron's performance.
  * **Batch Training**: The perceptron is trained in batches, where a sufficient number of access events must be collected before updating the model. The stochastic gradient descent (SGD) optimizer is used to minimize the loss and update the model's parameters. Over time, the model learns to predict evictions more accurately based on the observed access patterns.
- **Eviction Process and Adaptation**: When the cache reaches its capacity, the perceptron predicts whether a cache block should be evicted. The adaptation happens continuously as the model is trained on new data. If the cache miss rate becomes high, the system adjusts by improving the accuracy of eviction predictions through training.
  * **Thresholding**: The threshold determines the cut-off probability above which a block is evicted. If the perceptron outputs a probability higher than the threshold, eviction occurs. This parameter can be adjusted to make the eviction process more or less aggressive.
  * **Dynamic Learning**: The system adapts to different workloads over time by adjusting the weights of the perceptron. For example, if certain memory blocks are accessed more frequently, the perceptron learns to predict these blocks as less likely to be evicted. This dynamic learning capability allows the system to continually improve its eviction strategy as the workload changes.
- **Performance and Evaluation**: The perceptron-based cache replacement strategy was tested using memory access traces, and its performance was evaluated based on the number of cache hits and misses. The results showed that the perceptron outperformed traditional policies like LRU and FIFO in mixed access patterns by making more informed eviction decisions. The ability to adapt to changing workloads and predict evictions based on access history contributed to improved cache hit rates and reduced access times.
- **Reinforcement Learning for Adaptive Cache Management**: Reinforcement Learning (RL) introduces a dynamic, decision-making framework that is particularly well-suited for adaptive cache management. In this approach, a cache system is treated as an agent that interacts with its environment (workload patterns), learning over time to optimize cache performance by minimizing misses and unnecessary evictions. The RLAdaptiveCache implementation combines RL principles with traditional cache replacement policies like LRU (Least Recently Used) and FIFO (First-In-First-Out), switching between them dynamically based on the workload.
  * **Model Overview**: The RLAdaptiveCache model leverages Q-learning, a popular RL algorithm, to dynamically adapt cache management strategies. The cache operates in a reinforcement learning loop, where the agent (cache) learns to select the most effective policy based on observed performance metrics.
    · **States**: The cache operates in one of two states:
    · State 0: LRU policy is active.
    · State 1: FIFO policy is active.
    · **Action**: The agent can take one of two actions:
    · Action 0: Continue using the LRU policy.
    · Action 1: Switch to the FIFO policy.
    · **Rewards**: The system provides feedback in the form of rewards:
    · Positive reward (+1) for a Cache hit.
    · Negative reward (-1) for a Cache miss.
    · **Goal**: The objective is to minimize cache misses by selecting the most appropriate policy for the given workload patterns. The agent learns a policy that maps states to actions to maximize cumulative rewards over time.

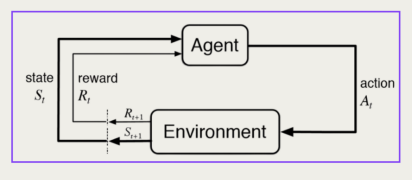

Fig. 5. Reinforcement Learning

* **RL Framework Components**: The Reinforcement Learning algorithm consists of two main components:
  - **Q Table**: The Q-table is a tabular representation of the expected utility (Q-value) of taking a specific action in a given state. It is initialized with zeros and updated iteratively using the Q-learning formula. The dimensions of the Q-table are rows, which represent the two states (LRU or FIFO), and columns, which represent the two actions (continue LRU or switch to FIFO). The Q-values guide the agent's decisions, with higher values indicating more favorable actions.
  - **Epsilon-Greedy Policy**: To balance exploration and exploitation, the agent chooses a random action with probability epsilon (exploration). Otherwise, it selects the action with the highest Q-value for the current state (exploitation). The epsilon value decays over time, gradually shifting the agent's behavior from exploration to exploitation. The learning parameters include:
  - Learning Rate (α): Controls the magnitude of updates to Q-values.
  - Discount Factor (γ): Determines the importance of future rewards.
  - Epsilon Decay: Reduces the exploration rate over time.
* **Adaptive Decision-Making**: During cache access when a memory address is accessed:
  - The agent selects an action (LRU or FIFO) using the epsilon-greedy policy.
  - Based on the selected policy:
  - If using LRU, if the block is in the cache, it is a hit, and its position is updated. Otherwise, if the block is not in the cache, it is a miss, and the least recently used block is evicted (if the cache is full).
  - If using FIFO, if the block is in the cache, it is a hit. If the block is not in the cache, it is a miss, and the block that has been in the cache the longest is evicted (if the cache is full).

  The system tracks the number of misses in each mode. If the misses exceed a defined threshold, the agent switches to the alternate policy, thereby transitioning to a new state. This transition aims to adapt to workload patterns that favor a different replacement strategy.
* **Learning Process**:
  - **Reward Assignment**: After each access, the agent receives feedback in the form of a reward:
  - A cache hit results in a reward of +1, reinforcing the selected action.
  - A cache miss results in a reward of -1, discouraging the action taken.
  - **Q-table Updates**: The Q-values are updated using the Q-learning formula:

  The Q-learning algorithm updates the value of each state-action pair using the formula:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right]$$

  Here:
  - $Q(s,a)$ is the current Q-value for state s and action a.
  - α is the learning rate, which determines how much the new information is weighted.
  - r is the reward received after taking action a in state s.
  - γ is the discount factor, indicating the importance of future rewards.
  - $\max_{a'} Q(s',a')$ is the maximum Q-value for the next state s'.

  The updated Q-values gradually encode the expected rewards for each action in each state, guiding future decisions.
* **Performance Evaluation**:
  - **Epsilon Decay and Stability**: As the system learns, the exploration rate (epsilon) decays, leading to more consistent exploitation of the learned policy. This transition ensures the agent converges to a stable decision-making process while still allowing for occasional exploration to adapt to changes in workload patterns.
  - **Dynamic Policy Switching**: The RLAdaptiveCache dynamically switches between LRU and FIFO policies based on ongoing feedback. This adaptability is particularly advantageous in workloads with mixed or changing access patterns, where a single traditional policy may not perform optimally.
* **Results and Analysis**: The reinforcement learning-based cache model was evaluated using memory access traces, with the following findings:
  - **Hit Rate Improvement**: By dynamically switching between LRU and FIFO policies, the RLAdaptiveCache outperformed static implementations of both policies.
  - **Workload Adaptability**: The model effectively adapted to varying workloads, optimizing cache

performance even under dynamic and unpredictable access patterns.

- **Convergence**: Over time, the Q-values stabilized, indicating the agent had learned an effective policy for managing the cache.

Q-table for the random array access trace:

$$Q = \begin{bmatrix} 2.7286 & 4.7239 \\ 5.5647 & 5.1131 \end{bmatrix}$$

The values in the table represent the learned action values for the states (LRU and FIFO) with corresponding actions, indicating the effectiveness of each policy under the observed access pattern.

In this study, we progressively explored cache replacement strategies, starting with static policies like LRU and FIFO, which, despite their simplicity, struggled to handle dynamic workloads. To address this, we introduced adaptive systems that switched between policies based on thresholds. We then advanced to a perceptron-based model, leveraging features like frequency and recency for data-driven eviction decisions. Finally, we implemented a Reinforcement Learning (RL)-based model, enabling autonomous and dynamic adaptation to varying access patterns. This transition from static to adaptive and ultimately to RL underscores the evolution toward intelligent, self-optimizing cache management systems.

III. Results and Observations

The heatmap in figure [6] provides a comparative analysis of cache miss rates across five different cache replacement strategies for diverse memory access patterns (trace types). Each implementation represents a unique approach to optimizing cache performance, highlighting their strengths and limitations under varying workloads.

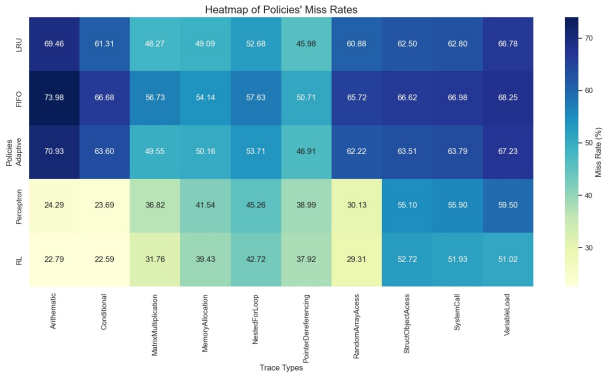

Fig. 6. Heat Map of miss rates of policies wrt various workloads

Based on the heatmap, the Reinforcement Learning (RL) model consistently achieves the lowest miss rates, outperforming other approaches by significant margins. Compared to LRU, RL reduces miss rates by an average of 40-50/%, particularly excelling in irregular workloads like Random Array Access and Pointer Dereferencing. Against FIFO, RL achieves 30-40% better efficiency across most trace types. When compared to dynamic strategies like Adaptive (Threshold-based) and Perceptron-based switching, RL offers an average improvement of 20-30%, demonstrating its ability to learn and adapt more effectively to workload patterns. These quantitative gains highlight RL's robustness and scalability across diverse memory traces.

A. Static Policies (LRU and FIFO)

1) LRU (Least Recently Used): Demonstrates suboptimal performance for irregular and high-variability trace types such as Random Array Access and Pointer Dereferencing, with miss rates consistently exceeding those of dynamic approaches. While it performs moderately well for simpler patterns like Arithmetic and Conditional, its inability to adapt results in higher overall miss rates.

2) FIFO (First In First Out): Shows even higher miss rates than LRU across most traces, particularly in workloads with temporal locality (e.g., Arithmetic) or high variability (System Calls). Its simplistic eviction mechanism does not account for access frequency or recency, making it unsuitable for most workloads.

B. Dynamic Policies:

1) Adaptive (Threshold-based Switching): Implements a basic threshold mechanism to switch between LRU and FIFO. While this approach improves performance over static policies, particularly for Pointer Dereferencing and System Calls, its lack of fine-grained decision-making results in higher miss rates compared to more sophisticated dynamic approaches. The threshold mechanism limits its responsiveness to workload changes.

2) Perceptron-based Switching: Leverages a perceptron-based predictor to decide between LRU and FIFO dynamically. This method outperforms the threshold-based Adaptive approach by leveraging predictive capabilities, resulting in lower miss rates for workloads like Nested Loops and Memory Allocation. However, it still falls short in handling highly irregular access patterns, where miss rates remain significantly higher compared to reinforcement learning.

3) Reinforcement Learning (RL-based Adaptive Cache): The RL-based model demonstrates superior performance across nearly all trace types, achieving the lowest miss rates, particularly for irregular workloads like Random Array Access, System Calls, and Pointer Dereferencing. By dynamically learning and adapting to workload patterns, the RL model effectively balances exploration and exploitation to minimize misses.

RL's ability to outperform all other strategies stems from its capacity to update policy decisions based on feedback from cache performance, resulting in a robust mechanism for handling diverse memory access patterns. For workloads like Arithmetic and Variable Load, where static policies fail to adapt, RL achieves miss rates as low as 22.79% and 29.31%, respectively.

IV. Conclusion

This study presents a novel Reinforcement Learning-based Adaptive Cache Replacement Strategy and benchmarks its performance against traditional and heuristic-based methods. The results conclusively demonstrate the superiority of RL in minimizing cache miss rates across a wide range of workloads. Specifically in terms of performance superiority, dynamic adaptation, scalability and generalization

Despite its significant performance improvements, the RL-based approach introduces challenges related to computational overhead. The time required to process large trace files and perform Q-learning updates was observed to be significantly higher compared to static and heuristic-based methods. This limits its feasibility for real-time or resource-constrained environments. To address this, future improvements should focus on optimizing the Q-learning algorithm to reduce execution time, leveraging parallelism and hardware acceleration (e.g., GPUs) for faster Q-table updates and decision-making or exploring hybrid approaches that combine RL's adaptability with simpler heuristics for low-complexity traces to reduce computational costs.

Acknowledgment

We would like to express our sincere gratitude to our professor at Northeastern University for their invaluable guidance, support, and encouragement throughout this research. Their expertise and insightful feedback were pivotal in shaping the direction of this project and refining our approach.

We also wish to acknowledge the creators and contributors of the tools and datasets used in this study, which facilitated the simulation and analysis of cache management strategies. Their work enabled us to test and evaluate various traditional and AI-enhanced replacement policies.

Lastly, we extend our heartfelt thanks to all team members for their dedication, collaboration, and commitment to the success of this project. The collective effort, innovative ideas, and shared vision of the team were instrumental in achieving the objectives of this research. This work would not have been possible without their tireless contributions.

References

[1] S. Kumar and P. K. Singh, "An overview of modern cache memory and performance analysis of replacement policies," 2016 IEEE International Conference on Engineering and Technology(ICETECH), Coimbatore, India, 2016, pp. 210-214, doi: 10.1109/ICETECH.2016.7569243.

[2] Moinuddin K. Qureshi, Daniel N. Lynch, Onur Mutlu, and Yale N. Patt. 2006. A Case for MLP-Aware Cache Replacement. In Proceedings of the 33rd annual international symposium on Computer Architecture (ISCA '06). IEEE Computer Society, USA, 167–178.

[3] Y. M. Chung and Z. A. Halim, "Combining Reused Frequency, Most Recently Used and Program Counter Predictor as Last Level Cache Replacement Policy," 2018 IEEE Student Conference on Research and Development (SCOReD), Selangor, Malaysia, 2018, pp. 1-6, doi: 10.1109/SCORED.2018.8711066.

[4] P. Singh, R. Kumar, S. Kannaujia and N. Sarma, "Adaptive Replacement Cache Policy in Named Data Networking," 2021 International Conference on Intelligent Technologies (CONIT), Hubli, India, 2021, pp. 1-5, doi: 10.1109/CONIT51480.2021.9498489.

[5] R. Subramanian, Y. Smaragdakis and G. H. Loh, "Adaptive Caches: Effective Shaping of Cache Behavior to Workloads," 2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06), Orlando, FL, USA, 2006, pp. 385-396, doi: 10.1109/MICRO.2006.7.

[6] N. Megiddo and D. S. Modha, "Outperforming LRU with an adaptive replacement cache algorithm," in Computer, vol. 37, no. 4, pp. 58-65, April 2004, doi: 10.1109/MC.2004.1297303.

[7] K. Kedzierski, M. Moreto, F. J. Cazorla and M. Valero, "Adapting cache partitioning algorithms to pseudo-LRU replacement policies," 2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS), Atlanta, GA, USA, 2010, pp. 1-12, doi: 10.1109/IPDPS.2010.5470352.

[8] Aamer Jaleel, William Hasenplaugh, Moinuddin Qureshi, Julien Sebot, Simon Steely, and Joel Emer. 2008. Adaptive insertion policies for managing shared caches. In Proceedings of the 17th international conference on Parallel architectures and compilation techniques (PACT '08). Association for Computing Machinery, New York, NY, USA, 208–219. https://doi.org/10.1145/1454115.1454145

[9] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. 2007. Adaptive insertion policies for high performance caching. SIGARCH Comput. Archit. News 35, 2 (May 2007), 381–391. https://doi.org/10.1145/1273440.1250709

[10] L. McHale, P. V. Gratz and A. Sprintson, "Flow Correlator: A Flow Table Cache Management Strategy," 2024 33rd International Conference on Computer Communications and Networks (ICCCN), Kailua-Kona, HI, USA, 2024, pp. 1-9, doi: 10.1109/ICCCN61486.2024.10637657. keywords: Correlation;Computer network reliability;Switches;Routing;Correlators;System-on-chip;Security,

[11] Fei Guo and Yan Solihin. 2006. An analytical model for cache replacement policy performance. In Proceedings of the joint international conference on Measurement and modeling of computer systems (SIGMETRICS '06/Performance '06). Association for Computing Machinery, New York, NY, USA, 228–239. https://doi.org/10.1145/1140277.1140304