CACHE REPLACEMENT INVOLVING TRADITIONAL AND RL TECHNIQUES

submitted in partial fulfillment of the requirements

for the degree of

BACHELOR OF TECHNOLOGY

in

COMPUTER SCIENCE AND ENGINEERING

by

DEEP MAHESHWARI CS18B008

KIRTIK JANGALE CS18B017

Supervisor(s)

Dr. Raghavendra Kanakagiri

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY TIRUPATI

MAY 2022

DECLARATION

We declare that this written submission represents our ideas in our own words and where others' ideas or words have been included, we have adequately cited and referenced the original sources. We also declare that we have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in our submission to the best of our knowledge. We understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

Place: Tirupati
Date: 27-05-2022

Deep Maheshwari
CS18B008

Kirtik Jangale
CS18B017

BONA FIDE CERTIFICATE

This is to certify that the report titled **CACHE REPLACEMENT INVOLVING TRADITIONAL AND RL TECHNIQUES**, submitted by **Deep Maheshwari, Kirtik Jangale**, to the Indian Institute of Technology, Tirupati, for the award of the degree of **Bachelor of Technology**, is a bona fide record of the project work done by him [or her] under our or [my] supervision. The contents of this report, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Place: Tirupati
Date: 27-05-2022

Dr. Raghavendra Kanakagiri
Guide
Assistant Professor
Department of Computer Science and Engineering
IIT Tirupati - 517501

ACKNOWLEDGMENTS

Foremost, we would like to express our sincere gratitude to our guide Dr. Raghavendra Kanakagiri for the continuous support, for his patience, motivation, enthusiasm, and immense knowledge.

Our sincere thanks also goes to Dr. Ajin George Joseph for helping us in using reinforcement learning in our project.

We would like to thank all the faculties who have taught us various courses throughout B.tech which helped a lot in project.

Finally, we would like to thank our families for their constant support throughout our life.

ABSTRACT

KEYWORDS: Cache Replacement; Reinforcement Learning; Reuse Distance.

Cache Replacement has been at the heart of computer architecture's problems to improve performance. A tremendous amount of research has gone into uplifting results close to the bound of optimal policy, which Belady's MIN gives. We do the literature review for modern cache replacement using reinforcement learning techniques and deep learning techniques. We began exploring our ways into this vast space of cache replacement by implementing a traditional approach involving the use of the SHIP table from the policy by Wu *et al.* (2011). This helped us to study the simulator and evaluate some metrics based on the traces of benchmarks we ran and understand the performance gain or loss. We study the paper by Sethumurugan *et al.* (2021). The work is done with the use of a Reinforcement Learning based technique called Q-learning combined with a Deep Neural Network to give insights into the selected features representing cache memory access patterns for a particular block in a set which involves *preuse distance, hits since insertion, LRU position, access type* to name a few. This simulator is discussed in detail in further chapters, where we talk about neural network and heatmap analysis. This simulator written in python is used to implement and run agents to train and test on a few sample traces. These traces are analyzed to pick a few features leading to the creation of a policy that we tried to reproduce on the Champsim, where we saw a decrement of around 4% in relation to baseline policy LRU. We were able to learn some insights from these that might help build a sound policy based on the quantification of access patterns and relate the weights of features to these. We present our future insights into these in chapter 3. We discuss the results and analysis in detail about these in chapter 4.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABBREVIATIONS

IPC	Instructions Per Cycle
PC	Program Counter
DQN	Deep Q-network
RL	Reinforcement learning
ML	Machine Learning
DL	Deep Learning
LLC	Last level cache
SHIP	Signature based hit predictor
LRU	Least Recently Used
LSTM	Long Short Term Memory
ISVM	Integer Support Vector Machine
RRIP	Re Reference Interval Prediction
RLR	Reinforcement Learned Replacement
PBHT	Predicted Bucket Hit Table
SHCT	Signature based hit count
RRPV	Re Reference Prediction Value

CHAPTER 1

INTRODUCTION

A cache is a smaller, faster memory situated near a processor core that stores copies of data from frequently accessed memory locations. Data is moved between memory and cache in fixed-sized units known as cache lines or cache blocks. When a cache line is transferred from memory into the cache, a cache entry is formed. The cache entry will include both the copied data and the requested memory address. When the CPU needs data, it immediately checks in cache memory whether it has data or not. If data is present, it results in a cache hit; else, if data is not in cache memory, it results in a cache miss. In case of a cache miss, the CPU retrieves data from the main memory and inserts a block of data into the cache.

As the cache memory size is limited, on a cache miss to make room for new entry, the cache may have to evict one of the existing entries. The replacement policy refers to the heuristic it chooses which entry to evict. The underlying challenge with any replacement strategy is that it must forecast which existing cache line is least likely to be used in the future. Because forecasting the future with 100% certainty is impossible, there is no ideal strategy for selecting among the various replacement plans available. Cache replacement policies specify different ways to evict an item from the cache with it is full. Some of the examples of cache replacement policies are FIFO (First In, First Out), in which the line that enters the cache first is evicted first, LRU (least recently used) in which line that has been unused for the longest time gets evicted, LFU (least frequently used) in which line which has least access count gets evicted. The best policy is Belady's MIN policy because it will always evict the line that will not be needed for the longest time in the future. But this is theoretical and can't be implemented in real-life since it is generally impossible to predict with certainty that exactly how far in the future information will be needed. Several static cache replacement algorithms have been developed, but they are restricted to a subset of access patterns and perform poorly in complex circumstances.

Cache replacement policies are continually evolving to achieve the theoretical outcomes outlined by Belady's MIN algorithm. The application of ML/DL/RL to cache replacement problems is becoming more popular. Machine learning has made significant advances in natural language processing, computer vision, audio recognition, and time series analysis. The use of these sophisticated and powerful technologies in computer architecture is not new and has been intensively researched to improve some areas such as branch prediction, cache replacement, and data prefetching. However, there are a few challenges to overcome when applying it to hardware predictors. Training a neural network involves massive amounts of data and resources, necessitating offline training. However, the offline model is less successful for an extensive range of computer programs with dynamic access patterns changes. Another critical difficulty is deploying the offline model on a hardware chip with limited memory. The model's prediction time can be a problem in time-critical applications. Apart from this, specific policies need hardware changes, which may incur additional costs. Nonetheless, machine-learning-based cache replacement algorithms beat static heuristics and may be regarded as a viable solution for improving overall system performance.

1.1 Objectives and Scope

Cache memory plays a very important role in improving overall system performance by reducing latency during program execution. The problem is to effectively manage a limited size cache, so minor cache misses occur. Our objective is to come up with a cache replacement policy for LLC[1] to improve the performance, and the scope is to analyze it using simulators. We used the Champsim simulator for heuristic-based policies and an open-source gym environment for implementing reinforcement-based cache replacement policies. We used the metrics like IPC(instructions per cycle), and LLC hit rate to compare our policy with other existing policies.

[1]The L1 and L2 caches are fixed with LRU policy simulators only give option to test policy on LLC

CHAPTER 2

Literature Review

Liu *et al.* (2020) cast cache replacement as an imitation learning problem, and the goal is to design a policy that will imitate Belady's policy. They trained a policy on an episodic Markov decision process. State at timestamp t, $s_t = (s_t^c, s_t^a, s_t^h)$ has 3 components. $s_t^a = (m_t, pc_t)$ is cache line address which is accessed at time t and program counter of instruction at time t. $s_t^c = (l_1, l_2, \ldots.., l_w)$ contains the address of cache lines which belong to the same set as accessed by s_t^a. $s_t^h = (m_1, m_2, \ldots., m_{t-1}, pc_1, pc_2, \ldots, pc_{t-1})$ is the history of all past cache accesses. Action at timestamp t is, If cache miss occurs, action $\in \{1, 2, 3, \ldots.., w\}$ where action w represents evicting line l_w . If a cache hit occurs, then no action a_{no-op} is used since no cache line needs to be evicted. Reward $R(s_t) = 0$ if cache miss otherwise 1. The goal of the policy is to maximize the total number of cache hits, $\Sigma R(s_i), i : [0, T]$. They used a machine learning model to learn the weights of each element in the state. Access history and current access are input to LSTM, which will output cell state and hidden state. For each line in the currently accessed cache set, the context for each line will be formed, which will depend on hidden states and the embedding of each line address in the set. A final dense layer with softmax activation will be applied to the context of each line. Then the policy will choose the action based on the values we get from the final layer for each line. During training, parameters of the model are updated based on the loss function, which calculates the loss value of the current state with respect to belady's policy.

Shi *et al.* (2019) formulated cache replacement as a sequence labeling problem where the goal is to label each access in a sequence with a binary label. The input is a sequence of loads identified by their PC, and the goal is to learn whether a PC tends to access lines that are cache-friendly or cache-averse. They chose to identify loads by their PC instead of their memory address because there are fewer unique PCs, so they repeat more frequently than memory addresses which speed up training. Another reason is size and learning time of LSTM both grow in proportion to the number of unique addresses, so identifying loads by memory address is not feasible. This paper proposed an attention-based LSTM learning model for cache replacement in an offline setting. Input to the model is the sequence of load instructions (PCs), and output is binary prediction to each element in a sequence indicating whether the corresponding load should be cached or not. They use a three-layer model. The first layer is the embedding layer, the second is 1-layer LSTM, and the third is the attention layer. From training the LSTM model in an offline setting, they derive three insights. First is model benefits from a long history of past PCs(particularly the past 30 PCs, after that no significant improvement). Second, the model can achieve good accuracy based on just a few source memory accesses. Third, Prediction accuracy is largely insensitive to the order of the sequence. Based on these insights, they propose a simple ISVM model which is easier to implement in hardware.

In the paper, Jain and Lin (2016) cache replacement algorithm can nonetheless learn from belady's MIN algorithm by applying it to past cache accesses to inform future cache replacement decisions. When applied to a memory-intensive subset of the SPEC06 benchmark, their solution improves performance over LRU by 8.4%. In this paper, they try to learn from belady's optimal policy called OPT and then generate a predictor trained to decide based on OPT policy. This predictor is named Hawkeye by the authors of the paper. The two main concerns involved in this approach are that it needs an efficient mechanism to regenerate OPT and the second is that it needs some large window of history to achieve certain accuracy. These problems are resolved by using liveliness intervals that capture both demand and reuse distances. The second problem is resolved by using set sampling.

The OPTGen part of the algorithm defines X to X' as the usage interval where X is the just past access of the same address as X'. It uses an occupancy vector to determine whether X' would be a cache hit based on the number of liveliness intervals overlapping if they are greater than the cache capacity. The updating of the occupancy vector is based on the decision made by OPTGen whether the current usage interval of X has all elements less than cache capacity then only it increments all the elements by one. The initial access of X is set to 0 always. The Hawkeye Predictor learns whether loads by a given instruction would have resulted in hits or misses under the OPT policy. If OPTgen determines that a line X would be a cache hit under the OPT policy, then the PC that last accessed X is trained positively; otherwise, the PC that accessed X last is trained negatively. Hawkeye first chooses to evict cache-averse lines, as identified by the Hawkeye Predictor. On every cache access (both hits and misses), the Hawkeye predictor generates a binary prediction to indicate whether the line is cache-friendly or cache-averse. This prediction is used to update the RRIP counter, which determines the relative priority of eviction of cache lines.

Sethumurugan *et al.* (2021) uses Deep Reinforcement Learning to learn a cache replacement policy. After analyzing the learned model few critical features that are going to impact the cache replacement decisions are extracted. The replacement policy RLR(Reinforcement Learned Replacement) is generated in this paper by using insights from the RL agent. This paper provides a systematic analysis of feature selection and the importance of feature selection. Selected features are used to form a policy and tested in the champsim simulator. On average, RLR sees improvements on single-core by 3.25% and four-core system performance by 4.86% over LRU, with an overhead of 16.75KB for 2MB last-level cache (LLC) and 67KB for 8MB LLC. This paper involves problem formulation of cache replacement in terms of reinforcement learning to analyze the practical implementation of the policy.

Problem formulation involves defining the state space, action space, and reward for the RL agent. The state-space involves some features picked which they thought as useful for decision in cache replacement like access type for line, set accesses, set accesses since misses, line pursue distances, LRU count for cache lines, hits since insertion, etc. The action is just the index of the cache line to be evicted. The reward is defined in such a way that it drives the agent to behave like belady's optimal policy. The agent is made up of a deep Q-network that involves a training mechanism via the MDP(Markov Decision Process). From the insights generated by the RL agent, the standout features of state-space were used to formulate a cache replacement policy(RLR). This RLR policy is just a formula involving priority calculation based on three major features selected, which were line preuse distance, last access type for the line, and hits from insertion till the current time step. This priority helps to perform eviction, and LRU bits are used to break the tie in case of multiple same eviction priorities. The weights assigned to the priority of preuse distance is eight while the rest others are 1. The problem with this static weights approach is that it doesn't adapt to varying cache access patterns. These problems and analysis of this paper are done in chapter 4 by us.

CHAPTER 3

Cache Replacement Techniques Implementation and Analysis

3.1 Our Modification to SHIP

The paper reffered is the classical one in cache replacement by Wu *et al.* (2011). We developed a policy based on extension of SHIP's idea whose results on few benchmarks are also presented below in terms of LLC access latency. While the complete statistics will be discussed in Chapter 4.

3.1.1 SHIP - Signature Based Hit Predictor

The signature is nothing but a hash generated from any entities we want to capture information about. In this case we want to predict re-reference behaviour of incoming cache line, so they used hash table to store signature based hit counts referring to basically how many times a particular block was accessed by address generated by that PC. This SHCT(signature based hit count) table is used to learn the RRPV(re-reference prediction values) values for every cache line. In simple words if the cache hit occurs we increase value of SHCT corresponding to current signature and set it's outcome bit to true, while if it's a miss then we check if it's outcome was false which means it wasn't used at all so we decrement the SHCT value for it's signature. Now we bring in the new block by replacing block with low RRPV value and set the RRPV of new block according to the new signatures value in SHCT. If SHCT is high we predict distant RRPV else we predict intermediate RRPV, where RRPV values range from 0-7 hence needing only 3 extra bits per entry for storage. Now to reduce storage in hardware set sampling is used. Since to develop policy only learning RRPV from few sets is sufficient hence we defer from using SHCT for all sets rather we create sample class for selecting few sets among all.

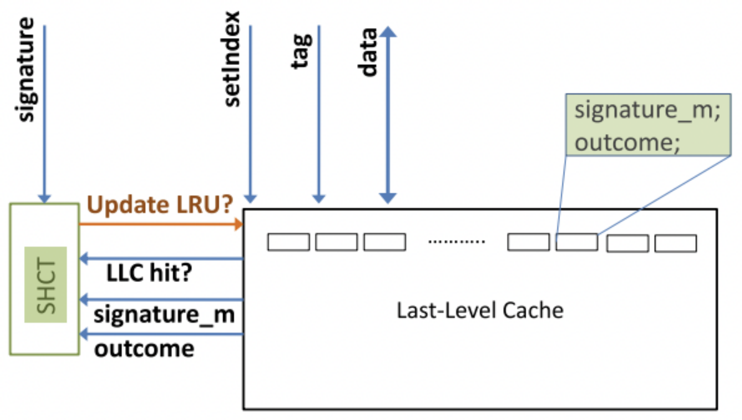


Figure 3.1: SHIP Implementation

3.1.2 Our policy inspired from SHIP

Now SHIP just retrieves data about re-reference from SHCT table from sampled sets. In our case we modify it a little bit, instead of keeping rrpv values table we maintain two entries per cache block. The one denoting frequency of reuse and the other if it is most recently used. The algorithm removes the block which is not in mru position and lowest reuse frequency. The SHCT table sets the reuse frequency based on count. If it's a hit, reuse frequency is set to 0 and mru of that is set to 1 and rest others' mru bit is set to 0. If it's a miss the reuse frequency of new block inserted is dependent on SHCT of the signature from which this block address was generated. In our case we use 3 bits for frequency counter and 3 bits for SHCT entry so we assign frequency equal to SHCT value itself when a new block comes.

Algorithm 1: Eviction of Victim

```
reused_val = 0
evict[LLC_WAY]
while true do
    count_reused = 0
    for way = 0 to LLC_WAYS {
      if reuse_frequency[S][way]==reused_val && MRU[S][way]==0 then
          evict[count_reused] = way
          count_reused++
    }
    if count_reused > 0 then
        return evict[rand()%count_reused]
    else
        reused_val++
    if reused_val==maxRRPV then
        return rand()%LLC_WAY
```

Algorithm 2: Updation of Reuse Frequencies based on SHCT

```
if hit then
    if reuse_frequency[s][b]<maxRRPV then
        reuse_frequency[s][b]++
    else
        for way = 0 to LLC_WAYS {
          reuse_frequency[s][way] /= 2
        }
else
    SHCT_idx = ip % SHCT_PRIME
      // SHCT_PRIME is prime number close to number of blocks and ip is program
      counter which generated address
    reuse_frequency[s][b] = 0
    if SHCT[cpu][SHCT_idx] > 0 then
        reuse_frequency[s][b] = 1
```

The following tables present the metrics IPC and LLC access latency(in terms of cycles) for LRU, Our policy and SHIP respectively.

Trace Name	LRU LLC_LAT	MY_REPL LLC_LAT	SHIP LLC_LAT
600.perlbench_s	153.945	153.826	150.618
602.gcc_s	282.53	292.932	276.362
605.mcf_s	131.008	138.561	120.571
607.cactuBSSN_s	77.3572	77.3455	77.2765
620.omnetpp_s	133.033	137.575	138.293

Table 3.1: Last level cache access latency for LRU, Our replacement policy and SHIP policy

CHAPTER 4

Reinforcement Learning driven cache replacement

4.1 The background and foundations of Reinforcement Learning

Reinforcement learning (RL) is growing as a subset of machine learning in which software agents take action and take action in the hope of maximizing high-priority rewards. There are several different forms of feedback that can determine how the RL system works. Compared to a supervised learning algorithm that maps a function from input to output, the RL algorithm usually does not include the target output (only the input is specified). The basic RL algorithm has three components: the agent (which can commit the action in its current state), the environment (which reacts to the action and provides the agent with new input), and the reward (the incentive or cumulative mechanism returned by environment). The basic scheme of the RL algorithm is shown below.

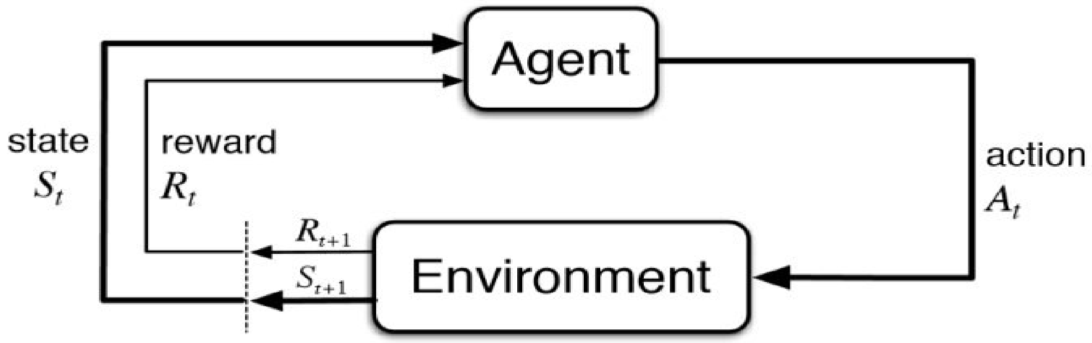


Figure 4.1: Schema Of RL

Definitions

- **Action (A)**: is defined as the possible moves that agent can take at this step
- **State (S)**: The situation in which the environment currently exists.
- **Reward (R)**: An immediate return send back from the environment to evaluate the last action.
- **Policy (π)**: Based on the current state whatever strategy is adopted by the agent is known as the policy.

- **Value (V)**: The expected long-term return with discount, as opposed to the short-term reward R. The expected long-term return of the current state sunder policy π is defined as Vπ(s).
- **Q-value or action-value (Q)**: Q-value is similar to Value, except that it takes an extra parameter, the current action a. Qπ(s, a) refers to the long-term return of the current state s, taking action a under policy π.

4.2 Deep Reinforcement Learning

In Deep Q-Learning, we utilise a deep neural network to approximate the ideal Q-function by estimating the Q-values for each state-action pair in a given environment. Deep Q-learning is the process of combining Q-learning with a deep neural network. Deep Q-Network, or DQN, is a deep neural network that approximates a Q-function.

Deep Q-Network: The network outputs estimated Q-values for each action that can be taken from a given state input. The goal of this network is to approximate the optimal Q-function.

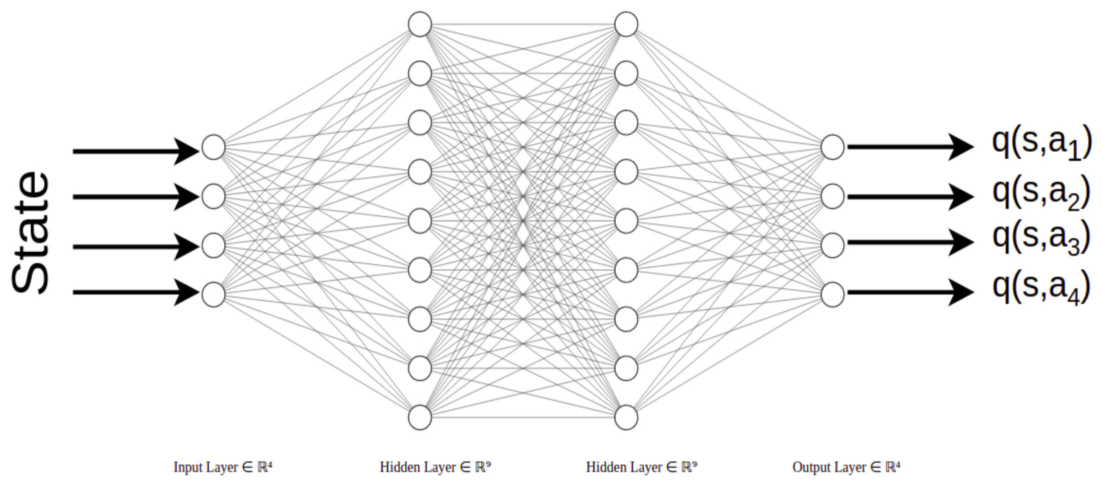


Figure 4.2: Deep Q-Network

How Deep Q-network is trained?

Experience replay: To train deep Q-networks we utilize a technique called *experience replay*. With experience replay, we store agent's experiences at each time step in list call called *replay memory*. At time step t, we define agent's experience at time step t as $e_t = (s_t, a_t, r_{t+1}, s_{t+1})$ where s_t is state of environment at time t, a_t is action taken by agent at time t, r_{t+1} is reward given to the agent at time $t+1$ and s_{t+1} is state of environment at time $t+1$.

Policy Network: We sample a random batch from replay memory at pass it to policy network. The model then outputs an estimated Q-value for each possible action from the given input state. Then loss is calculated by comparing Q-value output from Q-values given by target network.

Target Network: The target network is a clone of the policy network. Its weights are frozen with the original policy network's weights, and we update the weights in the target network to the policy network's new weights every certain amount of time steps.

Deep Q-Learning : Algorithm

Algorithm 3: Deep Q-learning with Experience Replay

```
Initialize replay memory $D$ to capacity $N$;
Initialize action-value function $Q$ with random weights;
for episode = 1, M do
    Initialize state $s_t$;
    for t = 1, N do
        With probability $\varepsilon$ select a random action $a_t$;
        Otherwise select $a_t = max_a\ Q^*(s_t, a; \theta)$;
        Execute action $a_t$ and observe reward $r_t$ and state $s_{t+1}$;
        Store transition $(s_t, a_t, r_t, s_{t+1})$ in $D$;
        Set $s_{t+1} = s_t$;
        Sample random minibatch of transitions $(s_t, a_t, r_t, s_{t+1})$ from $D$;
        $Set\ y_j = \left\{ \begin{array}{ll} r_j, & for\ terminal\ s_{t+1} \\ r_j + \gamma max_{a'} Q(s_{t+1}, a'; \theta), & for\ non-terminal\ s_{t+1} \end{array} \right\}$   (4.1)
        Perform a gradient descent step on $(y_j - Q(s_t, a_t; \theta))^2$;
```

4.3 Problem Formulation

Reinforcement Learning is a machine learning paradigm in which an agent tries to navigate through an environment by choosing an action from a set of allowed actions. Using the suggested action, the environment moves from the current state to the next state and generates a reward as feedback to the agent. The agent tries to maximize reward over a certain period of defined time. Q learning is one way of doing this via maintaining a state action table and applying the bellman optimization function overtime to get the maximum cumulative reward. However, it could be infeasible to implement such a table when the state and action space is enormous. In such a scenario, a neural network can be used as a function approximator in place of a table.

Since the effect of the action is Markov, RL has the potential to learn the theoretically optimal policy. RL is suitable for cache replacement problems because it adapts to dynamic changes in the environment and can handle the critical consequences of selected actions. Cache replacement is represented as a Markov decision process (MDP) in which the agent makes the replacement decision. Given the state of the cache, the replacement decision made by the agent causes the cache to move to the new state. Agents are rewarded based on how close the exchange decision is to BELADY (optimal). Our framework uses the RL algorithm to train neural networks and learn replacement policies. It analyzes the neural network and uses the insights gained from the neural network to derive a viable replacement algorithm for the hardware implementation. This section details the simulation framework and architecture exploration flow.

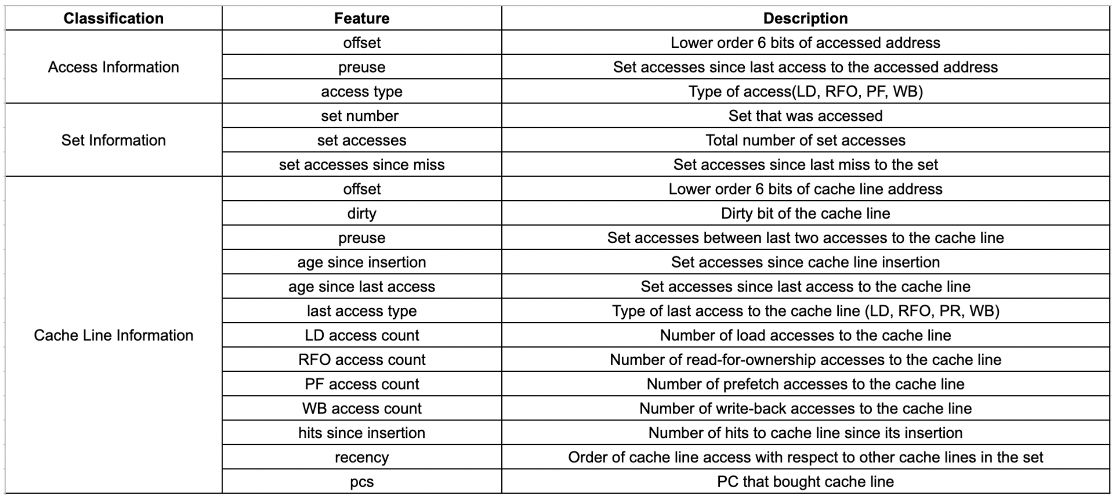
The simulation and testing consist of two parts, trace generation, and RL training. We use ChampSimISCA (2017) from the 2nd Cache Replacement Championship (CRC2) to generate LLC access traces. The trace file comprises a record of <PC, Access Type, Address> for each LLC access. Access types include load (LD), request for ownership (RFO), prefetch (PR), and writeback (WB). The trace is fed into a Python-based cache simulator taken from a gym environment provided by the Imitation Learning paper by Google, where we have implemented an RL agent inside modifying the state space. The cache simulator employs the same LLC setup as ChampSim and populates the LLC with the addresses that have been accessed. Each cache line is associated with a collection of cache states. In the diagram above, you can see the simulation framework. The cache simulator updates the cache state on a hit and moves on to the subsequent access. On a non-compulsory miss, the cache simulator interacts with the agent to make a replacement decision.

1. A state vector is given to the agent, containing information about the missed access and the accessed set.
2. For an n-way set-associative cache, the agent assesses the state vector and produces an output vector with n entries.
3. The value of each element in the output vector corresponds to a way in the cache

set, and it indicates how beneficial it is (from the agent's perspective) if that way is picked for eviction. The cache simulator then makes a replacement choice based on the output vector provided by the agent while also generating and sending a numerical reward to the agent for additional training.

4. Below we explain the critical components in detail.

State Vector: The information needed to make a replacement choice is included in the LLC state vector. We divided the LLC state into three types of information: a) access information, which represents the current cache access; b) set information, which describes the set being accessed; and c) cache line information, which describes each cache line in the accessed set. The statistics of the accessible set are updated with each LLC access. The set access counter, for example, is increased with each access to the set. For instance, on every hit to the set, the set access counter since miss is increased, and on a miss, it is reset to zero. Similar counters are maintained for every cache line in the set. In the case of an eviction, the cache line counts are reset to begin counting for the newly added cache line. Cache lines are additionally enhanced to hold additional information such as the last access date, the kind of previous access, the dirty bit, and other essential aspects. Table II contains the complete feature list for the LLC state. One-hot encoding is used for categorical information like the last access type. Access count is a numerical property normalized by its greatest value and displayed as a fractional number between 0 and 1. The feature offset is the lone exception, for which we utilize a 6-bit binary representation (assuming 64-byte cache lines). We use 334 floating-point values to represent a state vector for a 16-way set associative LLC.



Classification	Feature	Description
Access Information	offset	Lower order 6 bits of accessed address
	preuse	Set accesses since last access to the accessed address
	access type	Type of access(LD, RFO, PF, WB)
Set Information	set number	Set that was accessed
	set accesses	Total number of set accesses
	set accesses since miss	Set accesses since last miss to the set
Cache Line Information	offset	Lower order 6 bits of cache line address
	dirty	Dirty bit of the cache line
	preuse	Set accesses between last two accesses to the cache line
	age since insertion	Set accesses since cache line insertion
	age since last access	Set accesses since last access to the cache line
	last access type	Type of last access to the cache line (LD, RFO, PR, WB)
	LD access count	Number of load accesses to the cache line
	RFO access count	Number of read-for-ownership accesses to the cache line
	PF access count	Number of prefetch accesses to the cache line
	WB access count	Number of write-back accesses to the cache line
	hits since insertion	Number of hits to cache line since its insertion
	recency	Order of cache line access with respect to other cache lines in the set
	pcs	PC that bought cache line

Figure 4.3: Features required for agent

Agent and it’s architecture: The agent takes the feature vector obtained by running the routine of state to feature transformation. This feature vector is of size 248, which is passed to an agent which transforms this to output via a deep neural network to the size of 16, which is cache associativity. These values correspond to the eviction priority of each cache line. The one with a minimum value is considered for eviction. The agent contains a neural network of four layers, with the first input layer having 248 neurons, a second hidden layer having 64 neurons, a third hidden layer having 32 neurons, and finally, the 16 layer output. The activation function used is RELU based on experimentation performed with RELU, softmax, tanh, and sigmoid.

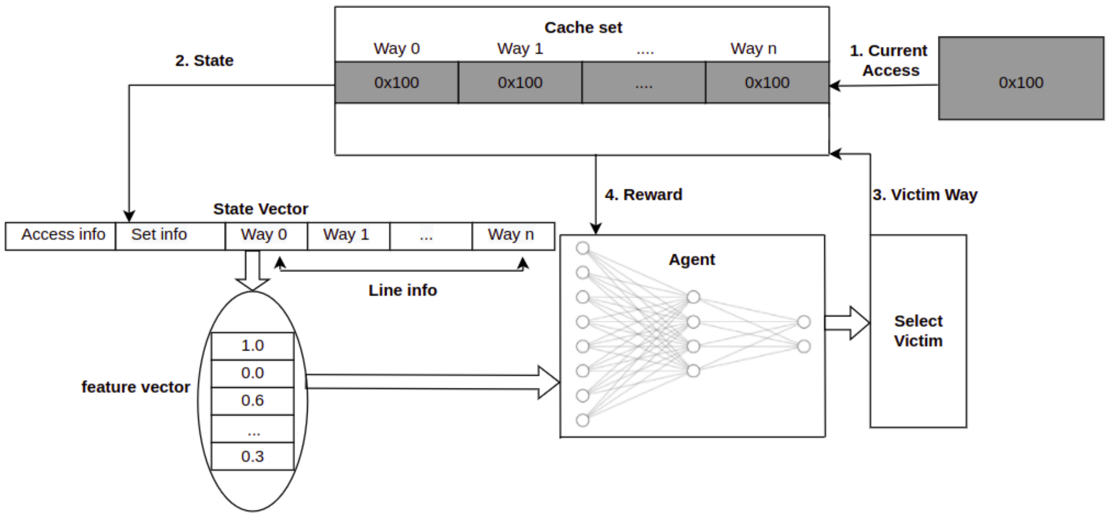
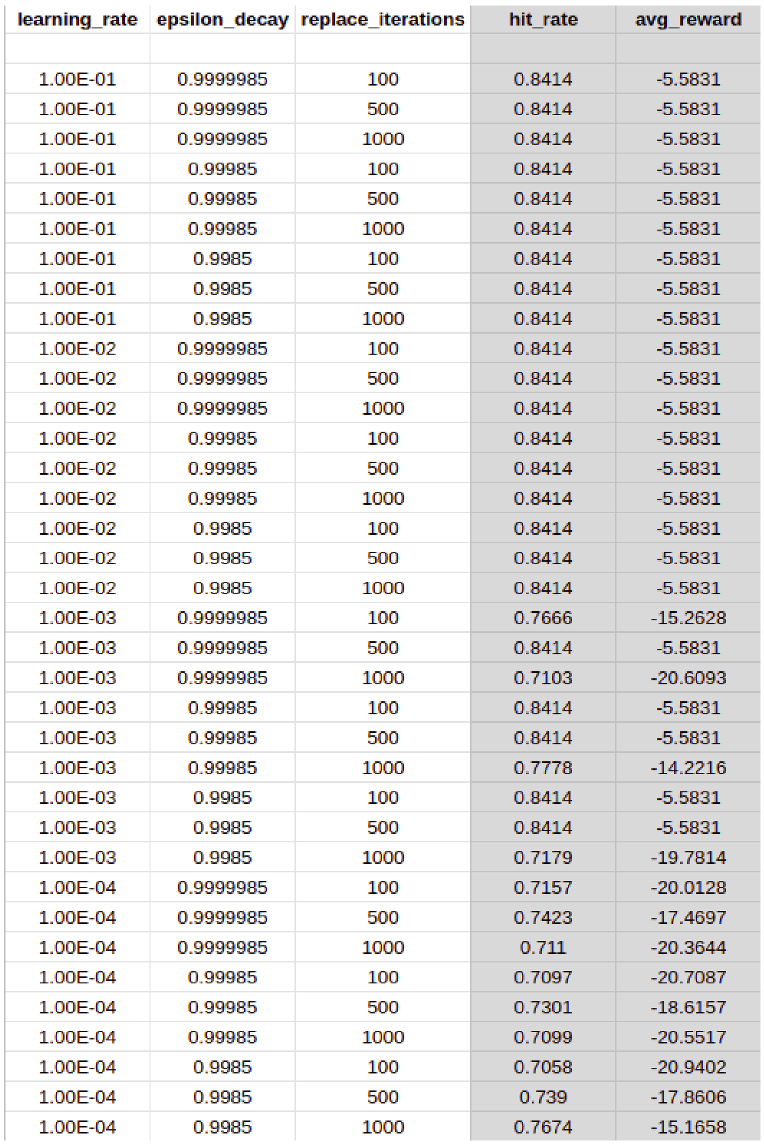


Figure 4.4: Simulation flow diagram

Replacement Decision: For each cache way a value is returned by agent which means that a n element vector is returned by agent for n-way associative cache (e.g. 16-element vector for a 16-way cache). The replacement decision is made by an ε greedy approach, in which we choose the victim with the maximum value with a probability of 1-ε and randomly select a victim with a probability of ε. Random actions explore new trajectories and expand the search space. In our training mode, we do epsilon decay starting from 1 with a decay factor of 0.99985.

Reward: The reward assigned is to make the agent learn how to behave like the optimal policy BELADY. Thus we have rewards assigned relative to reuse distances of cache lines. The positive reward is assigned if the cache line evicted is the same as belady’s decision. The neutral reward is assigned if the cache line evicted has a reuse distance more than the incoming cache line. For all other cases, a negative reward is assigned.

Training: For training, the mechanism explained in the previous section is used. An experience replay buffer is used. A neural network is trained using a batch of sampled tuples from the buffer. The training parameters were chosen by simply running through all combinations to get the best set of parameters corresponding to randomly selected traces. The below figure demonstrates hit rates for a trace with different combinations of parameters.



learning_rate	epsilon_decay	replace_iterations	hit_rate	avg_reward
1.00E-01	0.9999985	100	0.8414	-5.5831
1.00E-01	0.9999985	500	0.8414	-5.5831
1.00E-01	0.9999985	1000	0.8414	-5.5831
1.00E-01	0.99985	100	0.8414	-5.5831
1.00E-01	0.99985	500	0.8414	-5.5831
1.00E-01	0.99985	1000	0.8414	-5.5831
1.00E-01	0.9985	100	0.8414	-5.5831
1.00E-01	0.9985	500	0.8414	-5.5831
1.00E-01	0.9985	1000	0.8414	-5.5831
1.00E-02	0.9999985	100	0.8414	-5.5831
1.00E-02	0.9999985	500	0.8414	-5.5831
1.00E-02	0.9999985	1000	0.8414	-5.5831
1.00E-02	0.99985	100	0.8414	-5.5831
1.00E-02	0.99985	500	0.8414	-5.5831
1.00E-02	0.99985	1000	0.8414	-5.5831
1.00E-02	0.9985	100	0.8414	-5.5831
1.00E-02	0.9985	500	0.8414	-5.5831
1.00E-02	0.9985	1000	0.8414	-5.5831
1.00E-03	0.9999985	100	0.7666	-15.2628
1.00E-03	0.9999985	500	0.8414	-5.5831
1.00E-03	0.9999985	1000	0.7103	-20.6093
1.00E-03	0.99985	100	0.8414	-5.5831
1.00E-03	0.99985	500	0.8414	-5.5831
1.00E-03	0.99985	1000	0.7778	-14.2216
1.00E-03	0.9985	100	0.8414	-5.5831
1.00E-03	0.9985	500	0.8414	-5.5831
1.00E-03	0.9985	1000	0.7179	-19.7814
1.00E-04	0.9999985	100	0.7157	-20.0128
1.00E-04	0.9999985	500	0.7423	-17.4697
1.00E-04	0.9999985	1000	0.711	-20.3644
1.00E-04	0.99985	100	0.7097	-20.7087
1.00E-04	0.99985	500	0.7301	-18.6157
1.00E-04	0.99985	1000	0.7099	-20.5517
1.00E-04	0.9985	100	0.7058	-20.9402
1.00E-04	0.9985	500	0.739	-17.8606
1.00E-04	0.9985	1000	0.7674	-15.1658

Figure 4.5: Hit rate and Average reward comparison for different combination of parameters

4.4 Results and Analysis

- **Heatmap Analysis**: The analysis was performed based on the weights assigned by the neural network to each of the features assigned. From what we see for different traces, we trained agents, and it assigns weights differently for each trace based on the relevance of the feature corresponding to the trace. All the features related to set accesses are almost 0 as they are not relevant for replacement decisions. In contrast, we find some non-zero values that we can consider an imperfection in the agent. For trace *403.gcc*, we find PC the most relevant, followed by dirty bits with relative weights of 1 and 0.94, respectively. On the contrary, in trace *450.soplex*, we see that the PF access count has the highest weight, followed by hits since insertion. In the case of *470.lbm*, we see that almost all relevant features(excluding set access-based features) have the same weight, with the highest weight for dirty bits followed by writeback access count, which is logical as both are related. On average, the features that shine out are dirty bit, pcs, recency, preuse distance, and last access type.

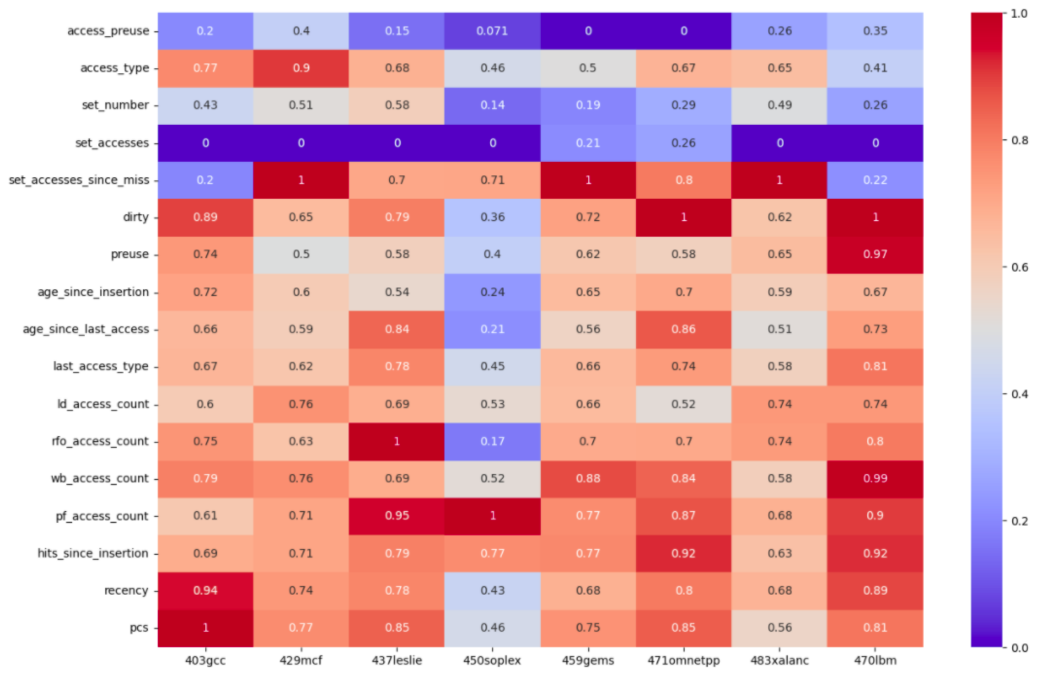


Figure 4.6: Heatmap Analysis

The heatmaps evolution for a single trace is also depicted in the figure as shown. The weights are being trained over time, and the relevant features get high weights. Over the last 100-300 iterations, we see the heatmap becomes stationary to particular weights.

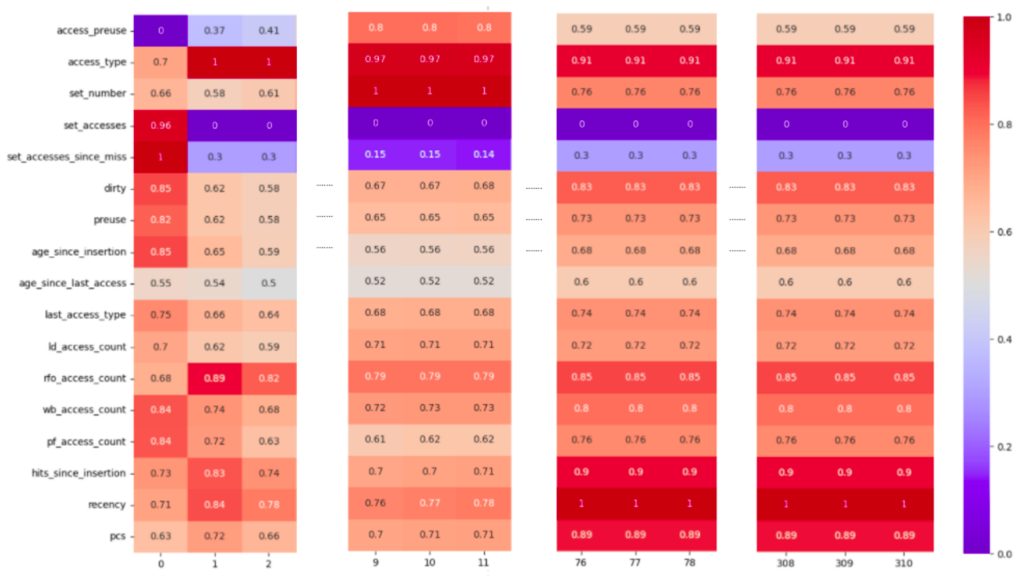


Figure 4.7: Heatmap Evolution

- **Statistics for different traces**: The hit rates for the different policies and our agent are plotted below. A different agent is used for every trace. So accordingly, the performances are recorded. We have train hit rates as well as test hit rates plotted.

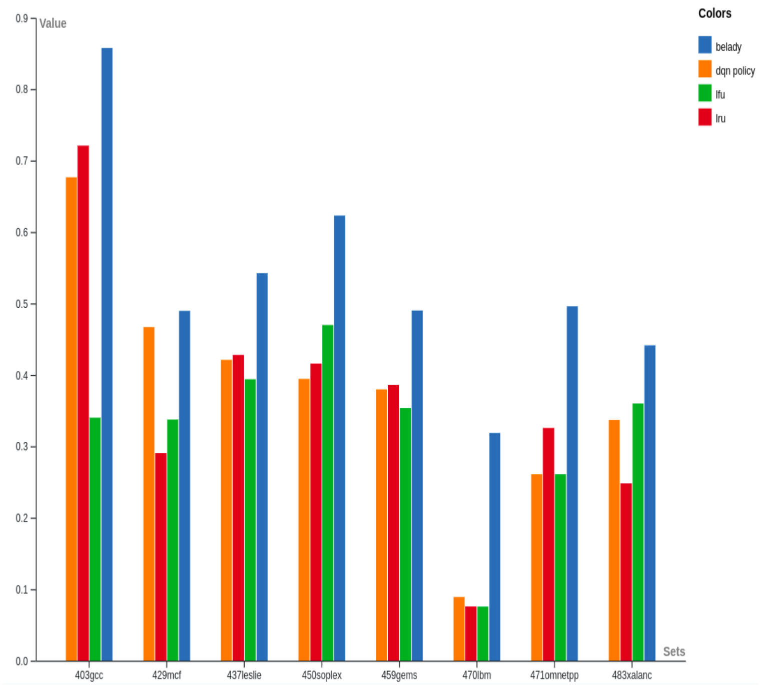


Figure 4.8: Train hit rates for different policies

The train hit rates show improvement over LRU in *429mcf*, *470lbm*, and *483xalanc*. In contrast, others are close or differ by a significant margin.

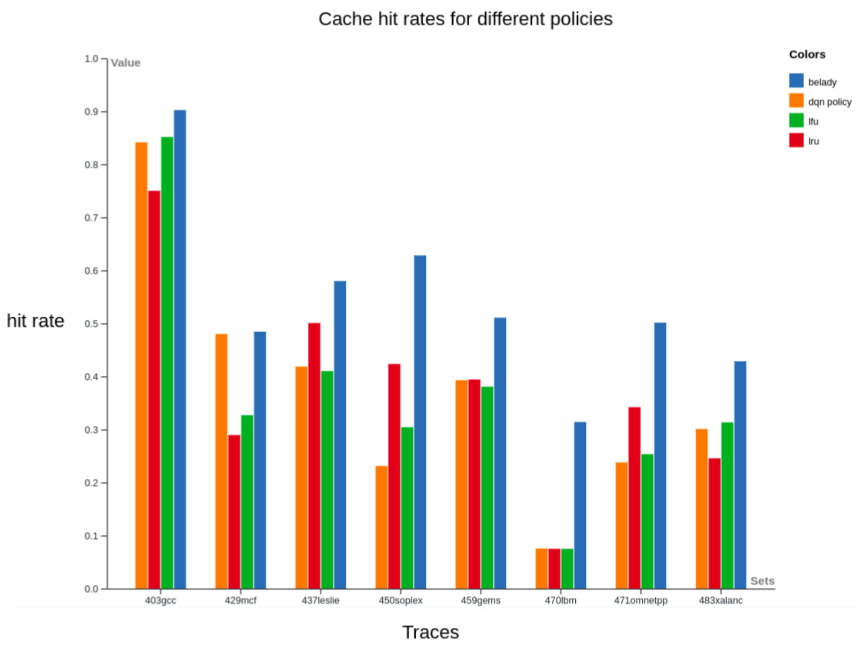


Figure 4.9: Test hit rates for different policies

In test statistics, hit rates of *429mcf, 459gems, 470lbm*, and *483xalanc* are close to or above LRU. The *429mcf* performs almost close to belady.

- **Statistics for individual feature**: We picked seven features: access preuse, line preuse, line access type, access type for incoming request, access counts, hits, recency, and pcs. We tested agents using these features for each trace to see their impact on replacement decisions. We plot the results below for each feature per trace.

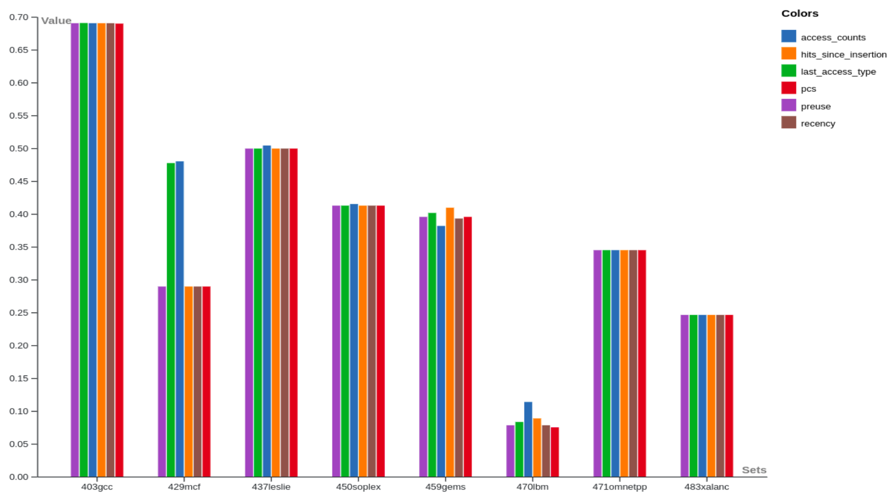


Figure 4.10: Hit rates feature wise for each trace

As we can see, there is no difference between the hit rates when features are used individually. So this experiment didn't let us infer much about feature importance.

- **Insights from the above all results and statistics**: The all of above results lead us to infer few of the following observations:-

- There are only a few relevant features that are important in the replacement decision to imitate belady.
- The features help in predicting reuse distance for a block in a way to imitate belady.
- Individually features have no value; only a linear combination with some weights helps derive a better policy.

4.5 Towards Practical Implementation on chip

The insights show that a few features are relevant in performing the cache replacement with varying importance for each trace. About the paper by Sethumurugan *et al.* (2021). The top features useful are preuse distance for each line, access type for each line, hits since insertion, and recency bits. While we could not reproduce the weights on an average, we propose to build upon the policy they have prepared from insights using these features. So below is the explanation of every feature and its usage to build up a policy:-

1. **Line Preuse**:- The line preuse denotes the past reuse distance of the line. The distance between current access and the last access to the particular line is known as preuse distance. The preuse distance of the line is related to reuse distance such that 90% of the cache line have $|\text{reuse distance - preuse distance}| < 50$. So as an estimate, we use preuse distance with some modifications as a proxy for reuse distance.
2. **Line Access Type**:- The access type of line has four categories, namely LD, RFO, PF, and WB. Based on the analysis done, most of the evictions by the agent were done early on for a line whenever PF was access type for that line.
3. **Hits since insertion**:- If the number of hits is zero, that line was evicted with a higher probability than any line with non-zero probability.
4. **Recency**:- The eviction of lines by the agent was inversely related to LRU. The one with the most recent access was removed over different traces than the one with the least recently used line.

According to the paper by Sethumurugan *et al.* (2021) the implementation involves a priority formulation as shown in the figure below:-

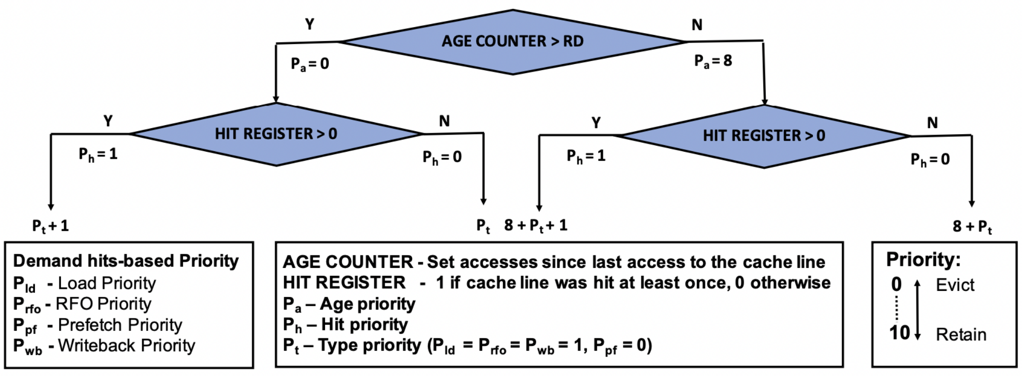

AGE COUNTER > RD
Y
N
$P_a = 0$
$P_a = 8$
HIT REGISTER > 0
HIT REGISTER > 0
Y
N
Y
N
$P_h = 1$
$P_h = 0$
$P_h = 1$
$P_h = 0$
$P_t + 1$
P_t
$8 + P_t + 1$
$8 + P_t$
Demand hits-based Priority
P_{ld} - Load Priority
P_{rfo} - RFO Priority
P_{pf} - Prefetch Priority
P_{wb} - Writeback Priority
AGE COUNTER - 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
10
Evict
Retain

Figure 4.11: Priority based formula based on above features

We propose to improve the above implementation of this given formula with static weights to rather dynamic weights. Based on the observed evolution of weights while training, we find out that DNN tries to adjust itself to the pattern over the first few iterations of an access pattern and stabilizes its weights for the next few 1000 iterations it stabilizes its weights. Hence, this leads us to infer that many different access patterns over trace lead to dynamic changes in priority for weights of each feature used. In the above figure, we can see that a weight of 8 is assigned to preuse distance, and weights of 1 are assigned to both hits since insertion and last access type. In case of a tie-break, recency bits are used.

The idea is to quantify access patterns somehow, i.e., spatial locality and temporal locality, and use them to adjust the weights of the features selected. Some of the previous techniques to quantify locality have been Weinberg *et al.* (2005) and Anghel *et al.* (2013).

4.6 Hardware Budget

Hardware Overhead of RLR

Each cache line has

- 2 bit Age Counter.
- 1 bit Hit Register.

So total 4 bits of overhead per cache line.

For $2MB$ 16 way LLC with $64B$ cache line, total storage overhead of RLR is $16.75KB$.

Hardware overhead of modified SHIP

In SHiP, 14 bit for *signature* and 1 bit for *outcome* is required for SHCT to learn the reuse pattern for the signature. Along with that modified SHIP, we used 3 bits for *frequency counter* and 1 bit for *mru* position for each line. So, a total of 19 bits of overhead per cache line. We used the set sampling technique in which we used 256 sets for 2MB 16-way associative LLC. So, the total storage overhead for modified SHiP is $4KB$.

4.7 Further Literature review and ideation for novel cache replacement using traditional techniques

After doing all the analysis on the simulator by the paper Liu *et al.* (2020) we figured out some insights that we discussed in the previous section. The insights make us believe that if we want to choose some heuristics for policies, their weights need to be learned on the go rather than assigning them static weights for a full run on the benchmark. This work has been best done by a paper Teran *et al.* (2016) which uses the perceptron model of learning weights on the go for various important features picked. This is a lightweight, hardware-friendly method adopted in place of a hard-core ML algorithm. We rather digress here to analyze a few more policies in reuse distance-based cache replacement rather than these heuristics-based cache replacement.

One such paper by Keramidas *et al.* (2007) is used to emphasize the use of reuse distance and ETA(Expected Time of Arrival). The reuse-distance of an address is defined as the number of intervening events, a notion of time between two consecutive references to this address. They use this PBHT (Predictor Bucket Hit Table) to save the confidence counters for reuse distance in each bucket which is log_2 of the reuse distance. Two major operations can be performed on this table one is update, and the other one is lookup. The lookup takes the PC and checks if it is already present in the table. The update operation update performs the update of reuse distance for a live PC that already exists in the table. They manage the replacement by calculating the ETA. The ETA's are calculated using the timestamps stored and the reused distance corresponding to the PC entry in PBHT that brought this block into the cache.

From this paper, we found an inspiration to implement our policy with the RDP table and ETA, which we have done ideation for and implemented to some extent. Interestingly, a paper that has bought breakthrough into cache replacement in terms of results is yet to be published and is also based on similar lines. This paper has a different training way of training and calculating the ETA.

4.8 Fairness in multi-core Prefetching

We, along with our co-team for BTP, also did work on the problem for multi-core prefetching, where we tried to implement a fairness model to allow changes in acceptance rates of prefetchers for each core based on their utility. For running a multi-core-based model, we need to create a mix of traces known as workloads. So we analyzed some traces and categorized them as prefetch friendly, LRU replacement friendly, etc. Now we take two benchmarks within which this classification was done. There are ten workloads, each created within these benchmarks. Then there are around ten workloads created across benchmarks.

Now the model is created as follows:- we define two things utility and acceptance rates. To modify the acceptance rate, the interval is defined as a number of prefetches issued which is currently set to 100. The number of prefetches in the past used to consider the retraining model is 1000. So within an interval of 100, we get to do the retraining of the acceptance rate. The large interval calculates relative IPC change and the percentage of prefetches accepted for that particular core. We define utility as the $\%\ prefetches\ accepted \times \Delta(IPC)$. Now, this utility is used to calculate acceptance rates of all cores based on the proportion of utilities that it gets times the total requested prefetches. Now, if these exceed the requested prefetches, the acceptance rate of that core is put to 100%. We have a few statistics of how speedup is achieved for different workloads over the run with no fairness, which we call the baseline model. These statistics are discussed in the following chapter.

Trace1	Trace2	Trace3	Trace 4
602.gcc-s0	623.xalancbmk-s1	607.cactuBSSN-s3	607.cactuBSSN-s0
605.mcf-s1	605.mcf-s4	654.roms-s5	619.lbm-s3
623.xalancbmk-s2	649.fotonik3d-s1	621.wrf-s3	607.cactuBSSN-s0
621.wrf-s3	654.roms-s5	607.cactuBSSN-s0	605.mcf-s4
607.cactuBSSN-s3	602.gcc-s2	623.xalancbmk-s2	607.cactuBSSN-s
605.mcf-s5	605.mcf-s4	649.fotonik3d-s3	607.cactuBSSN-s3
605.mcf-s1	607.cactuBSSN-s3	649.fotonik3d-s3	619.lbm-s3
607.cactuBSSN-s0	602.gcc-s0	649.fotonik3d-s1	605.mcf-s3
605.mcf-s1	605.mcf-s5	623.xalancbmk-s1	649.fotonik3d-s3
623.xalancbmk-s2	605.mcf-s4	607.cactuBSSN-s0	649.fotonik3d-s
pr-10	cc-6	pr-3	bc-12
bfs-14	bfs-10	cc-13	cc-6
pr-10	bfs-10	bc-3	bc-12
bc-3	bc-12	pr-10	bfs-10
pr-3	bc-12	bc-3	bfs-14
bc-3	bfs-14	cc-13	bfs-10
pr-3	bc-12	pr-10	cc-6
bc-3	bc-12	pr-3	bfs-14
pr-10	cc-6	cc-13	bfs-10
bfs-14	bc-12	pr-3	cc-6
649.fotonik3d-s3	607.cactuBSSN-s0	bfs-10	bc-3
654.roms-s3	605.mcf-s1	cc-6	pr-10
621.wrf-s3	cc-13	cc-6	654.roms-s5
605.mcf-s3	607.cactuBSSN-s3	pr-3	bf
619.lbm-s3	bc-3	bc-12	649.fotonik3d
623.xalancbmk-s2	623.xalancbmk-s1	bfs-10	pr-10
bc-12	cc-6	605.mcf-s1	602.gcc-s2
605.mcf-s4	619.lbm-s3	pr-3	cc-13
623.xalancbmk-s1	607.cactuBSSN-s3	pr-10	bc-12
619.lbm-s0	605.mcf-s5	cc-13	bfs-14

Table 4.1: Mixes created with traces from different benchmarks

CHAPTER 5

Results and Discussions

We started exploring some of the techniques from the reviewed literature and the simulator on which the policy for cache replacement needed to be implemented. We present our discussion on the results mentioned in the above respective chapters.

The extension over the SHIP policy we implemented the stats are as mentioned below.

For overall SPEC17 traces, which we have taken for the testing, we see the geometric mean of 0.5081, while for LRU, it comes to about 0.5041, so we see a 0.7% improvement, but which is of no significance. Also, compared to SHIP with LRU policy, there is, in fact, a degradation of 2% as we see a geometric mean of 0.5182 for SHIP. The average hit rate of our SHIP-based policy is 0.42. While for the original SHIP policy, it is 0.43. which is 1% decrement. While comparing to LRU, which has a hit rate of 0.38, we have about 4% gain.

For overall GAP traces, which we have taken for the testing, we see the geometric mean of IPC for our SHIP policy which is 0.267, a speedup of a very insignificant amount compared to LRU. The original SHIP policy has a geometric mean of IPC of 0.277 compared to the LRU policy, which is 0.267. The average hit rates are 0.347, 0.354, and 0.382, respectively, for LRU, our SHIP-based policy, and the existing SHIP-based policy.

The plots for a few sample traces have been plotted, showing the IPC speedup over LRU in both the benchmarks SPEC17 and GAP.

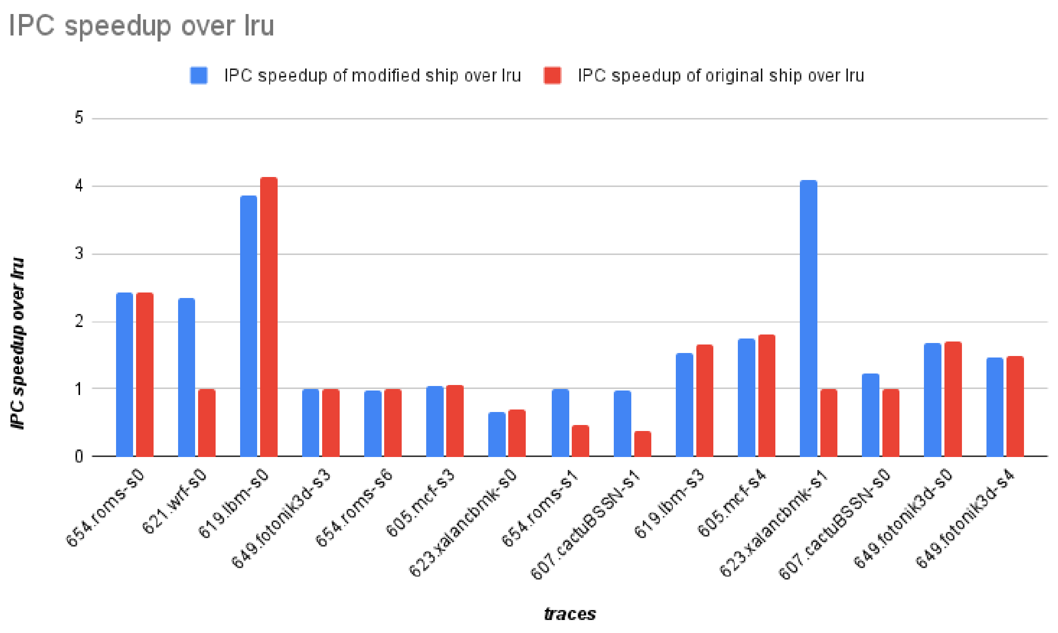


Figure 5.1: IPC speedup of our SHIP based policy v/s existing SHIP based policy over LRU in some SPEC17 traces

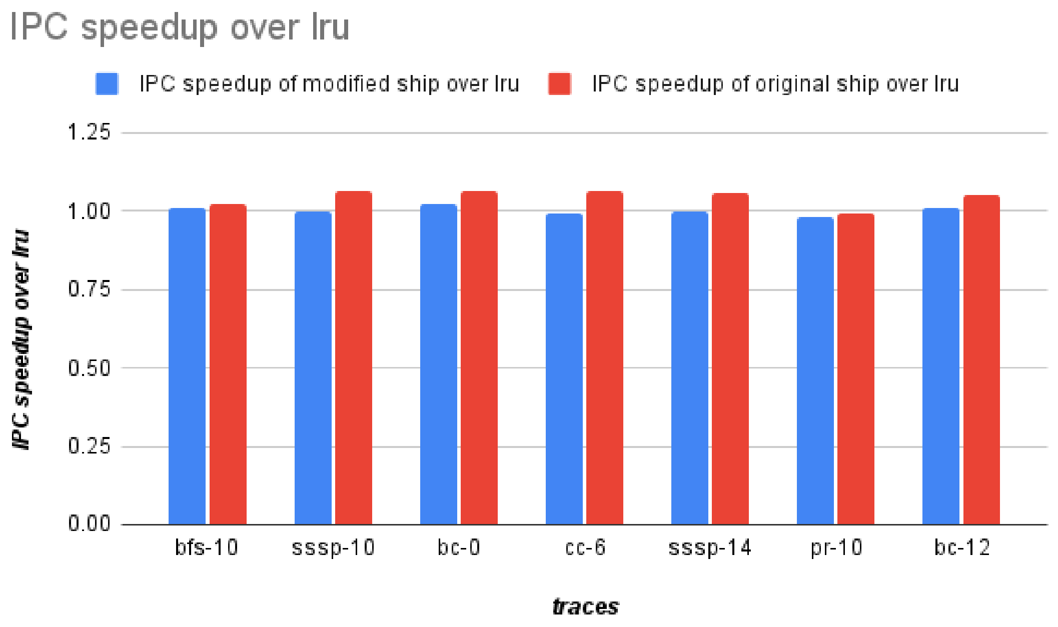


Figure 5.2: IPC speedup of our SHIP based policy v/s existing SHIP based policy over LRU in some GAP traces

We have also done the practical implementation of the policy RLR in Sethumurugan *et al.* (2021) The paper below presents some speedup stats for sampled traces in SPEC17 and GAP benchmarks.

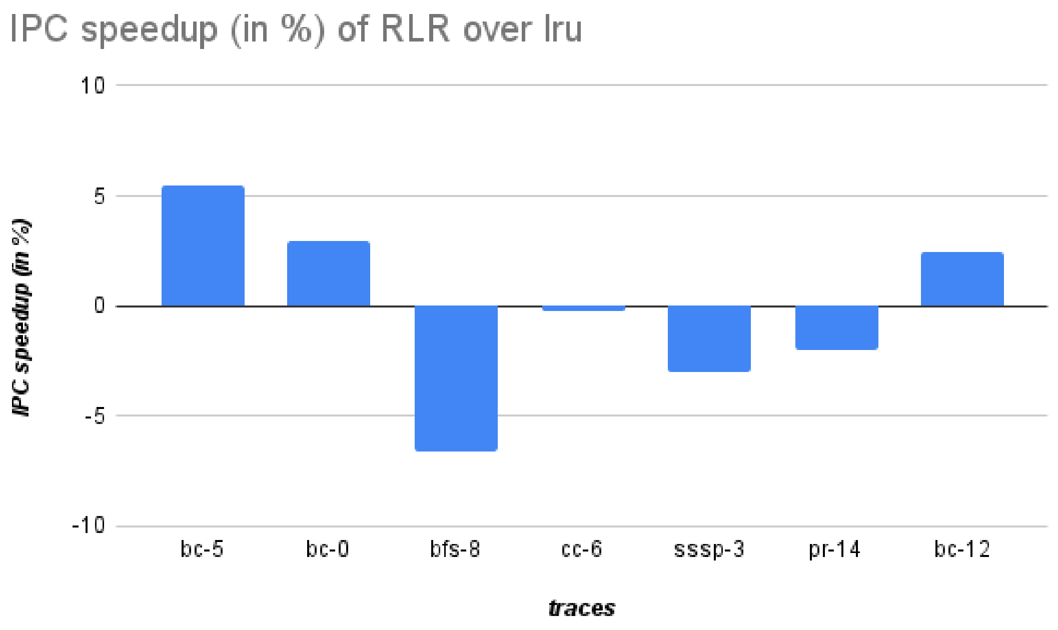


Figure 5.3: IPC speedup of RLR over LRU in some GAP traces

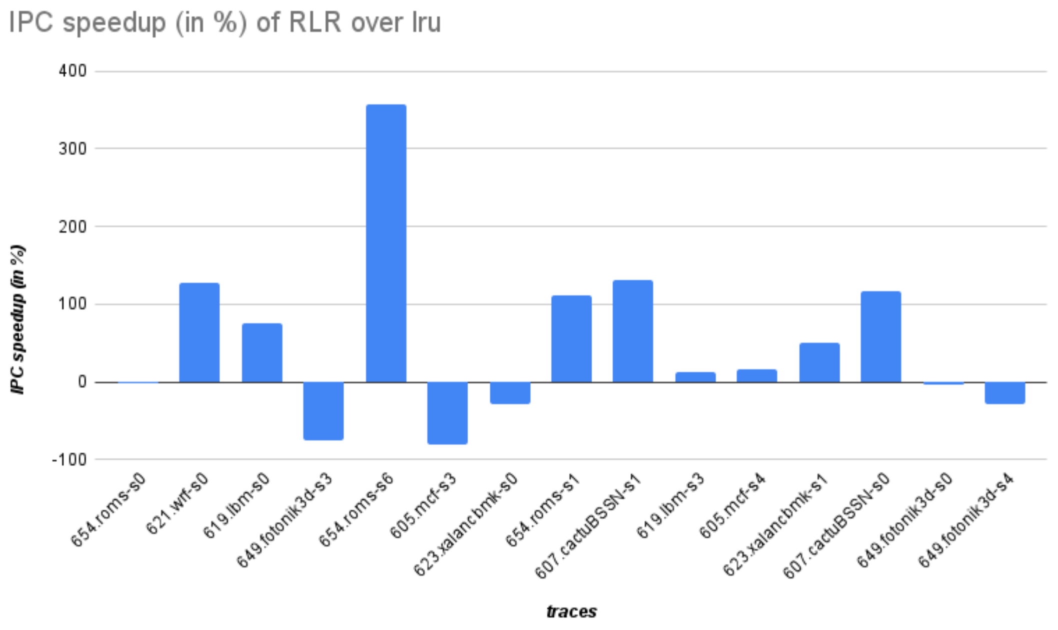


Figure 5.4: IPC speedup of RLR over LRU in some SPEC 17 traces

So on average, we see most of the traces have significantly achieved a speedup over LRU policy. The geometric mean of IPC speedup with respect to LRU is 0.96, which means a 4% drop for SPEC17 traces in performance, implying our results are not intact with the results given by the paper. Though the traces individually seem to be performing great in terms of high speedups achieved, for example, in 654.roms-s6 there is almost 400% increase while in case of 654.roms-s1, 607.cactuBSSN-s1, 621.wrf-s0 all of them have achieved a speedup of above 100%. In the case of GAP traces, we see a speedup of limited magnitude compared to SPEC17 traces. For instance, trace bc-5 has a speedup of around 6% while bfs-8 has a down gradation of about -6%, so overall in GAP, not much improvement is seen in terms of IPC on an average.

Policy	Prefetcher	Average hit rate
SHIP	-	0.4327
Modified SHIP	-	0.4212
LRU	-	0.3851
RLR	next line	0.7016
LRU	next line	0.70816

Table 5.1: Average hit rates on SPEC17 traces

Policy	Prefetcher	Average hit rate
SHIP	-	0.3822
Modified SHIP	-	0.3541
LRU	-	0.3473
RLR	next line	0.5075
LRU	next line	0.5142

Table 5.2: Average hit rates on GAP traces

On average, we see that hit rates remain the same almost across three policies without prefetching. The policies with prefetching have a gain in hit rate although the relative gain with respect to LRU with prefetching for RLR with prefetching is still negative as seen from the table. The modified SHIP and SHIP approximately dominate LRU by 4-5% in hit rates.

5.1 Discussion of stats in multicore prefetching with fairness

We define two main speedups for calculating improvements over baseline.

- Harmonic speedup which is defined as HS = $\frac{N}{\sum_{i=0}^{N-1} \frac{IPC_i^{alone}}{IPC_i^{together}}}$
- Weighted speedup which is defined as WS = $\sum_{i=0}^{N-1} \frac{IPC_i^{together}}{IPC_i^{alone}}$

Based on the stats collected we present the speedups for the given 30 mixes.

Version	Geomean of Harmonic Speedup	Geomean of Weighted Speedup
Preference to core 0	0.939	0.935
Preference to core 1	0.942	0.936
Preference to core 2	0.936	0.948
Preference to core 3	0.936	0.936
fairness model	0.952	0.959

Table 5.3: Geomean of IPC speedup over different cases

Based on the above stats we infer that model is not performing fair enough to give overall better speedup than no fairness. It is still below by like 5% approximately.

CHAPTER 6

SUMMARY AND CONCLUSION

The cache replacement domain has been evolving for a long time, with the first few policies being on a recency-based heuristic. To quote a few of them, LRU(Least Recently Used) is the most popularly used policy. It is the baseline policy to which the new algorithms are compared to check performance gains. The few other popular policies that are based on recency are TLRU(Time aware LRU), SLRU(Segmented LRU), and MRU(Most recently used). Some frequency-based policies have existed in traditional mechanisms of cache replacement. The Least Frequently Used(LFU) is one of the most popularly known policy under this heuristic. Some of the other policies can be named Least Frequently Recently Used(LFRU) and LFUDA(LFU with dynamic aging). The traditional algorithms have tried to exploit the relationship between these heuristics to the optimal cache replacement strategy.

We started with the literature review involving various papers involving modern analysis techniques for cache replacement in contrast to the traditional techniques mentioned above. The modern techniques involve the use of deep learning and reinforcement learning. The learning techniques have not been able to go on the hardware yet. The analysis was obtained from running the learning-based approaches for cache replacement or predicting the reuse distance for replacement. We went on to go for implementation of basic policy building up over the SHIP policy. The policy relies on SHCT(Signature-based hit count) table, a PC indexed table for confidence values of re-reference prediction. This heuristic gives significant improvement over baseline LRU. While we tried to use this SHCT table implementation from SHIP and implement our policy with reuse frequency and mru position bit, we could see 0.7% of improvement over LRU for overall traces we ran and tested on. Still, there were no gains over existing SHIP implementation with LRU.

The implementation of this traditional policy was mainly meant to get familiar with research in cache replacement involving the extensive use of the simulator, performance metrics analysis, and refining or tuning parameters. The Champsim simulator is the one in which we implemented the algorithm. This simulator gives a lot of interfaces like cache, dram, memory controller, etc. The simulator runs on traces that are available as a benchmark. This help evaluates the performance over different memory access patterns. We learned about the different utility functions given for implementing the policy. From there on, we explored modern techniques of cache replacement.

We studied a few modern approaches to cache replacement, mainly involving deep learning and reinforcement learning. The various papers in the reviewed literature did some offline analysis and got the insights to implement policy in hardware. We went through one paper by Zhong *et al.* (2018) related to cache replacement using RL. We took out the code template foundation from the GitHub source for this. Finally, along the lines of our thought in deep Q learning, we got a paper recently published in HPCA by Subhash et al. We figured out a way to implement the algorithm. We chose a simulator from an imitation learning paper provided by google research. We set up the simulator to involve state space and all other things required for an agent to function.

Finally, after implementing the agent on a few sample traces, we could analyze a heatmap. Unfortunately, we couldn't replicate the results to create some standout features. We couldn't get hit rates similar to the one in the paper. The one possible reason for this is that there is no standard simulator for cache replacement in python. From this analysis, the policy framed was also implemented in Champsim, and those were also not successfully reproduced. We concluded our Btech Project by analyzing the stats and presenting them in chapter 5. At last we also did a small research under area of multi-core cache prefetching which is still under progress.

REFERENCES

1. **A. Anghel**, **G. Dittmann**, **R. Jongerius**, and **R. Luijten**, Spatio-temporal locality characterization. *In 1st Workshop on Near-Data Processing (WoNDP)*. 2013.

2. **ISCA** (2017). The 2nd cache replacement championship **Champsim**. URL https://github.com/ChampSim/ChampSim.

3. **A. Jain** and **C. Lin**, Back to the future: Leveraging belady's algorithm for improved cache replacement. *In 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 2016.

4. **G. Keramidas**, **P. Petoumenos**, and **S. Kaxiras**, Cache replacement based on reuse-distance prediction. *In 2007 25th International Conference on Computer Design*. IEEE, 2007.

5. **E. Liu**, **M. Hashemi**, **K. Swersky**, **P. Ranganathan**, and **J. Ahn**, An imitation learning approach for cache replacement. *In* **H. D. III** and **A. Singh** (eds.), *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*. PMLR, 2020. URL https://proceedings.mlr.press/v119/liu20f.html.

6. **S. Sethumurugan**, **J. Yin**, and **J. Sartori**, Designing a cost-effective cache replacement policy using machine learning. *In 2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2021.

7. **Z. Shi**, **X. Huang**, **A. Jain**, and **C. Lin**, Applying deep learning to the cache replacement problem. MICRO '52. Association for Computing Machinery, New York, NY, USA, 2019. ISBN 9781450369381. URL https://doi.org/10.1145/3352460.3358319.

8. **E. Teran**, **Z. Wang**, and **D. A. Jiménez**, Perceptron learning for reuse prediction. *In 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016.

9. **J. Weinberg**, **M. O. McCracken**, **E. Strohmaier**, and **A. Snavely**, Quantifying locality in the memory access patterns of hpc applications. *In SC'05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*. IEEE, 2005.

10. **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 2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2011.

11. **C. Zhong**, **M. C. Gursoy**, and **S. Velipasalar**, A deep reinforcement learning-based framework for content caching. *In 2018 52nd Annual Conference on Information Sciences and Systems (CISS)*. IEEE, 2018.