Deep Reinforcement Learning Based Cache Replacement Policy

Patel Mit , 2020csm1016
Indian Institute of Technology Ropar
2020csm1016@iitrpr.ac.in

A. Problem Statement

Despite extensive advances in the development of deployment rates, designing a repository reserve policy that mimics Belady's (Optimal) algorithm remains a challenging task. This paper will implement the Deep Reinforcement Learning based cache replacement policy, which learns from Belady's algorithm by using past cache access history to efficiently predict future cache references based on Belady's Algorithm. I have demonstrated that this implementation can accurately make predictions when simulating Belady's behaviour to improve cache replacements. using the Standard SPEC2006 benchmark suite through the generating Memory Trace using ChampSim and USIMM simulator and will compare the results to other existing cache replacement policies.

B. Introduction

The Caches are an important part of today's Processor ,The performance of the Cache is largely influenced by its Replacement Policy. Effective replacement policy can effectively reduce off-chip bandwidth usage to improve the performance of the whole system. There is a big body of the previous work of archive change policies; However, designing savings repository policies still exists a challenge for chip designers.

Commonly Least Recently Used (LRU) as well Re-Reference Interval Prediction (RRIP) policies are used as replacement Policy refresh reference counters. However, these are driven by static heuristic policies that only apply to the limited cache category access patterns only. Using Program Counter (PC) information, Modern evolving policies are able to capture changes in the change in the access cache patterns, and can effectively reduce MPKI in a wider spectrum of replacement Policy, The final most Efficient Policy is Belady(Optimal). But Belady is the correct theory; we can not implement it directly. Undoubtedly, PC-based policies (Hawkeye, Glider etc) work very well with non-PC (LRU , FIFO, DRRIP etc) policies in almost all cases.

The use of these very modern and powerful design tools Reinforcement Learning based Deep Neural networks is not a new Idea. This Idea has already been researched to improve specific areas such as the branch prediction, repositioning data , Prefetcher Etc. However, there are few challenges when it comes to using it in hardware predictions and Implementation in actual Hardware and Neural Training the network needs large amounts of data and resources to Train therefore offline training is required. But in general computer programs are flexible changes indicated by access patterns so using the offline model proves that It did not work as well as It would in online Training. Apart from this, other methods require hardware conversion and may cause more up. However, machineLearning based cache replacement policies are more effective than ever static heuristics and can be considered a possible solution to improve overall system performance and performance measuring. In this case paper, we are exploring some recent cache replacement policy based on machine learning-based policies and expor a new Deep Reinforcement Learning based approach to improve overall performance of cache by designing a better replacement policy which will mirror the Beledy's (optimal) Algorithm.

C. **Related Work**

a. **Hawkeye:** It uses a PC-based prediction to find out if the cache line is cache-friendly or cache averse. By default, the policy starts with selecting cache-averse lines. If no cache-averse cache lines are found in the set, the old repository line accessed is selected for extraction. For SPEC 2006 CPU benchmarks, Hawkeye reduces miss rates over LRU by 8.4% with 2MB LLC and by 15% for four core systems with 8MB LLC

b. **Glider:** It uses a Vector Support Machine (SVM) hardware predictor based on repository. Initially, short-term memory-based memory (LSTM) the model is used to improve predictive accuracy. Then, the authors translate the offline model for details and hand create a feature that represents the control flow history of the system. Then, a simple sequential learning model is used for simulation Predictive accuracy of LSM. Hardware implementation Policy requires Register Opposition History Program and a complete SVM table.For SPEC 2006 and 2017 programs, Glider reduces the miss rate by 8.9% over LRU in a single core configuration, whereas Hawkeye only reduces it by 6.5 percent. Glider reduces the miss rate over LRU by 14.7 percent in a multicore scenario, whereas Hawkeye reduces it by 13.6 percent.

c. **LeCAR:** It is an ML-based repositioning algorithm designed for smaller cache sizes that uses the benefits offered by the known static heuristics, LRU and LFU . All caches are provided for any of the policies based on possible distribution Weights for LRU and LFU. The LeCAR framework is modeling a learning problem of consolidation and reward minimization. The basic principle of minimizing reward. rewards refers to the right course of action accepted. LeCAR maintains the FIFO line that holds the latest release (history) in the cache. All entries in line written policy that led to its release i.e., LRU or LFU. If the generated reference is available online in history, its policy-related rewards increase with weights another policy updated showing the best the decision would have been made. It’s an online model where the model learns after missing everything to reduce rewards.

d. **Reinforcement Learning Replacement (RLR):** While Hawkeye and Glider are both very active PC-based forecasts, it is important to have cost effective switching techniques with small overhead and hardware development. In the RLR, initially, a set of targeted features were obtained by training a learning reinforcement (RL) agent as well hill analysis. Features use distance, line the last type of access, line hits from the installation and the latest line are selected based on the weights of the neural network. The selection process was completely automatic, again allowing the RL agent to adapt to the changing variables' access patterns. Then an exchange policy was developed using these limited features by providing essential levels of warehouse lines. The PC was deliberately removed from this feature set as it adds to the weight of the hardware once further communication occurs in transferring PC data to LLC. RLR works much better than non-PC support DRRIP forecasts. With careless overhead, the RLR improves single-core and four-core system performance at 3.25 and 4.86 percent, respectively, over LRU.

Problems with Hawkeye ,Glider and Many other policies which are based on PC(program counter) Unfortunately, installing a PC is a switch policy not only does it require additional overhead of storage but it also involves significant changes in processor and data management. PC access to LLC requires a streaming PC at all levels of cache hierarchy, including extensions data path, to modify the cache structure to save a PC, adds additional PC storage to the Trouble Line, Reset Buffer (ROB) and Load / Store Line

(LSQ), and more. The more expensive there is the more work to implement PC-based policies that require repair of the entire processor pipeline.So PC based Solution is not very useful.

LeCAR is designed for smaller cache so it will not be helpful for LLCC and it is essentially modeling which policy is relatively better LRU or LFU so any way if we get better accuracy still it is not enough to get as accurate as Beledy (optimal) policy.

RLR(Reinforcement Learning Replacement) is Training in offline mode so it can not adopt Agent behaviors while running in processor so it has less hit rate as compared to training online because online is Dynamic it can adopt the nature of Agent as per recent memory references.

D. Implementation

Our solution will enhance the approach used in the Reinforcement Learning Replacement (RLR) Paper which used offline Learning as Experiment shows offline Learning has Less Accuracy. Instead we will make a Reinforcement learning based Replacement Policy using online Training Deep Q Neural Network with more accuracy and light weight by adding only important features.

Our solution has two major parts: first if History Buffer which stores cache Access History and second Agent which is trained to Predict which block needs to Replace.

A. Input (State) Vector Space:

The LLC Input(state) vector contains the required information to make a decision instead. We have divided the LLC region into three categories of factors:

1. access information that describes the current access to the Cache
2. set the information describing an accessible set

In our basic model, we use frequency as the features of states.The motivation to use frequency information is that we observe the temporality of data access as well as the good adaptability of LFU, as we can see in the section . But keeping all-time access times requires memory cost of O(N) which is extremely high. Therefore, we only keep a small subset of history instead and adopt a neural network to regress and explore the latent information from it. This method reaches high effectiveness and avoids high memory cost.

B. Agent

Traditional Q-Learning updates Q-values Q(s, a) iteratively after experiencing a transition. Due to the huge state space and action space, we cannot use the traditional Q-Learning to store every Q-value in a hash table. Instead, we exploit feature-based Q-Learning, approximate Q-Learning. The Q-values are computed by feeding real-time state features s and applying function fθ : s → Q ∈ R^|A|, where each element Q(a) of Q is the Q-value Q(s, a) by taking action a. By exploring the environment, suppose the DRL agent experiences a new transition (s, a, s0) and gains reward r, the Q-values can be updated on-the-fly through these observations. The target Q-values Qk (s, a) is calculated from the current Q-values Qk−1(s, a)

$$Q'_k(s,a) = r + \gamma \max_{a'} Q_{k-1}(s',a')$$

where γ is known as the discount factor in reinforcement learning, k is the number of parameter updates. If we update θ at T0 period, T0 · k = t, where t is the current decision epoch, The traditional Q-Learning chooses to update Q-values by optimizing the temporal difference error (a.k.a. TD difference) between current Q-values and Artificial Intelligence target Q-values. But with the feature representation, the update on Q-values can only be done on the parameters θ. One of the most popular methods to optimize the temporal difference is gradient descent. We adopt the squared form of TD difference as the loss function, formulated by Equation .

$$\mathcal{L} = \|Q'_k - Q_k\|^2$$

MLP is more effective and widely-used to solve regression and prediction problems. Since we prefer value-based methods, this learning framework is also called Deep Q-Network (DQN).

C. Victim block selection

Output vector of the Deep Neural network (The Agent) returns the n vector n values, one for each cache line (e.g. size 64 vector for 64-way cache). The decision will be decided by ε Probability approach, when selecting a victim with a maximum value among 64 is (1 - ε)and chances to randomly select the victim is ε. To avoid local Maxima of performance

D. Reward Policy

The award directs the agent to further study appropriate replacement policy, so rewards policy work should be selected carefully. A replacement policy with a better idea, such as Belady(Optimal), It will replace a long-range reusable cache line with the distance between the lines per set. Allowing the agent to read this behavior, a good reward is returned when the agent does good resolution and pulls the cache line very far from the reuse range. Negative reward is refunded if the agent releases a cache line that has a lower reuse rate than the repository cached line.

We define the most basic reward of transition (s, a, s0)

$$r(s, a, s') = H(s, a, s') + \lambda P(s, a, s') \cdot \mathbf{1}(a \text{ causes miss})$$

where H(s, a, s0) is the reward term and P(s, a, s0) is the penalty term. λ balances the proportion between two terms. 1(*) = 1 if * takes place, otherwise 1(*) = 0. The reward term is supposed to be large when the selected action is judicious and results in relatively high hit count before the next decision epoch. Hence, it is defined as a weighted sum of hit count on each cache slot, formulated

$$H(s, a, s') = \sum_{i}^{C} w_i(h_i(t+1) - h_i(t)) = \boldsymbol{w}(\boldsymbol{h}(t+1) - \boldsymbol{h}(t))$$

where h(t) is the histogram vector of cumulative hit count of each cached content at t-th decision epoch. We employ a weight w here to highlight differences on each cache slot. For instance, we can increase the weight on newly cached items to endorse the latest selected actions. As for the penalty term, it is supposed to bring a reduction to the final reward when the next cache miss is exactly caused by the previously taken action. It should also hold the property that the penalty decreases as the hit count accumulated between two decision epochs

E. Online Training

For online we have to design a Beledy(optimal) algorithm which gives an optimal Cache line. In offline Training we already have a memory trace so we can easily design algorithms for optimal(Beledy) cache lines. I have designed a Deep Q Network for ML Agent for each SET of the cache line.

For the first method we can design a memory buffer which stores info of victim block Eg. We have Memory sequence A, B , C, D , B , D , C , E , C , A . First it has frame size 3 so once it comes to block D at TimeStamp 4 Controller has to select the Victim block.

Case 1: Let's assume it select A as victim then we will store Frame [A,B,C] with TimeStamp 4 in buffer once we reach to A at TimeStamp 10 we will search last access time stamp of all block of frame [A=10 , B=5, C=9] Highest time is 10 for 'A' and actual prediction was also 'A' in that case we train model with positive Rewards.

Case 2: Let's assume it select C as victim then we will store Frame [A,B,C] in buffer once we reach to C at TimeStamp 10 we will search last access time stamp of all block of frame [A=1, B=5, C=9] Highest time is 9 but time Stamp of A is even Less then TimeStamp of Frame(4) that means our Prediction was wrong in that case we train model with negative Rewards.

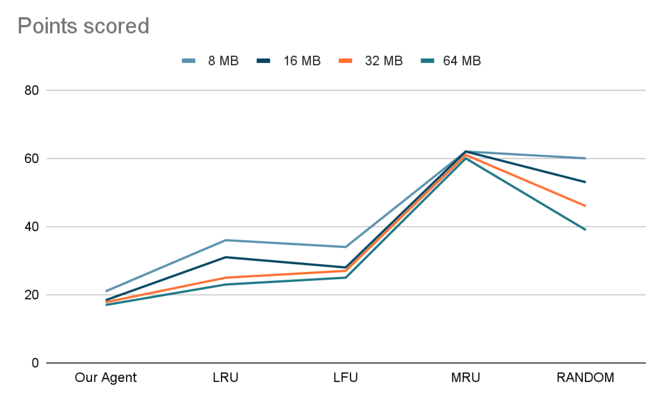
E. Experiments

In order to evaluate our method's performance, I have designed two experiments for miss rate assessment and one experiment for model tuning so we can improve our machine learning Model. As comparisons, I also evaluate the performance of FIFO, LRU, LFU, MRU, Clock and Random in the first three experiments.

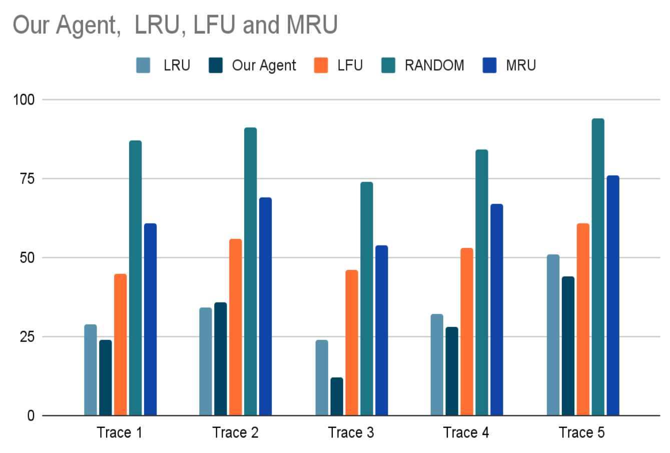
The dataset in our experiment has (.csv) file. The dataset used for simulation is derived from ChampSim and USIMM trace and then converted into Deep-Q-Network Trace (.CSV) format. I have then used this trace to train and simulate the Functional Simulation for miss rate.
We will use below two experiments to Evaluate our Solution.

1. I use the simulation data memory trace in this experiment. We will use 5 different traces which are generated by SPEC benchmark. We have actually used a small subset of benchmarks. And study HIT Rate of Various Replacement Policy including LFU Deep-Q-network , LRU etc.

2. In the second Experiment I have taken one Trace which is generated by SPEC benchmark and then Simulated with Different cache size. And study HIT Rate of Various Replacement Policy including LFU Deep-Q-network , LRU etc. for different Cache Size.



F. Results :as we can see in the first graph our Deep-Q-Network agent has average least miss rate for all cache Memory Size and MRU has Highest Miss rate. For Random and LFU(least Frequently) policy Miss Rate is decreasing with increasing Cache Size .Foe Deep-Q-Network Agent increasing in Cache size is not effecting Miss rate that means DQN agent is already performing optimal with small Cache size which is very good achievement because it is performing as good in 8 MB cache and 64 MB . .



In the second Graph we can see that I have compair Miss rate of various Replacement policies for five different Memory Access Traces which are derived from Champsim and USIMM. Our Deep-Q-Network Agent is performing better in all Trace except in Trace 2. It has better performance than the LRU.Random policy has the highest Miss rate followed by MRU policy. Throw this experiment we can say that DQN agent is better then other in majority case

G. Conclusion

In this report, I have proposed and implemented a reinforcement learning based cache replacement strategy by online learning. I trained our agent via a Deep Q Network and evaluated its performance under simulation operating system environments. Compared with LRU, LFU , MRU , RANDOM and other traditional replacement policies, our method produces a substantial hit rate in Majority memory trace and has very good adaptability in Extreme Memory Trace. My future work will mainly focus on Correctness and the effectiveness of the proposed method in real operation systems with very Extreme large Memory Trace. Due to limited time I was not able to Experiment it on a large Memory Trace. Additionally, My Current Study is only Focused on the single thread it is requesting.in Future i will try to Study on multiple Parallel Thread. For that I would either design a new thread context with corresponding learning model or refine our learning framework to be adaptive to other complex models.