Proceedings of the Thirty-Second International Conference on Automated Planning and Scheduling (ICAPS 2022)

An End-to-End Automatic Cache Replacement Policy Using Deep Reinforcement Learning

Yang Zhou, Fang Wang, Zhan Shi, Dan Feng

Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology
Luoyu Road 1037, Wuhan, China
{zhouyang1024, wangfang, zshi, dfeng}@hust.edu.cn

Abstract

In the past few decades, much research has been conducted on the design of cache replacement policies. Prior work frequently relies on manually-engineered heuristics to capture the most common cache access patterns, or predict the reuse distance and try to identify the blocks that are either cache-friendly or cache-averse. Researchers are now applying recent advances in machine learning to guide cache replacement policy, augmenting or replacing traditional heuristics and data structures. However, most existing approaches depend on the certain environment which restricted their application, e.g, most of the approaches only consider the on-chip cache consisting of program counters (PCs). Moreover, those approaches with attractive hit rates are usually unable to deal with modern irregular workloads, due to the limited feature used. In contrast, we propose a pervasive cache replacement framework to automatically learn the relationship between the probability distribution of different replacement policies and workload distribution by using deep reinforcement learning. We train an end-to-end cache replacement policy only on the past requested address through two simple and stable cache replacement policies. Furthermore, the overall framework can be easily plugged into any scenario that requires cache. Our simulation results on 8 production storage traces run against 3 different cache configurations confirm that the proposed cache replacement policy is effective and outperforms several state-of-the-art approaches.

Introduction

The cache replacement policy is to study the selection of blocks in the cache to replace under certain conditions. Designing a high-performance cache replacement policy suitable for various scenarios is still a challenging and time-consuming task. In most cases, the cache size is much smaller than the content of the workload, and limited space will greatly affect the hit rate of the cache. Cidon et al. (2016) show that improving cache hit rates of web-scale applications by just 1% can decrease total latency by as much as 35%.

As two classic cache replacement policies, LRU (Least Recently Used) and LFU (Least Frequently Used) are widely used because of their simplicity and stability. LRU policy and its variants base their replacement decision on the recency of references, while LFU policy and its variants base their decision on the frequency of references. To inherit the benefits of the two policies and allow a flexible trade-off between recency and frequency of references in basing the replacement decision, LRFU (Least Recently/Frequently Used) proposed by Lee et al. (2001) establishes a unified analysis equation for the two policies and adjusts the parameters in the equation so that LRFU can determine the ratio of recency and frequency of references as needed. Megiddo and Modha (2003) propose a self-adjusting, low-overhead, and efficient cache replacement policy ARC (Adaptive Replacement Cache). By simply dividing the access frequency into single and multiple times, ARC realizes a rough combination of recency and frequency. Park and Park (2017) propose a policy named FRD (frequency and reuse distance). FRD utilizes both the access frequency and reuse distance of a block to determine which blocks should remain in the cache. Recently, the machine learning (ML) research trend expands to the system performance optimization field, most followed the idea of intuitions and heuristics. Inspired by *ensemble learning*, Ari et al. (2002) use LRU, LFU, and FIFO (First in First Out) cache replacement policies to vote the blocks to be replaced in the cache, and adjust their weights according to the hit rate of each policy. They call this method ACME (Adaptive Caching using Multiple Experts). ACME presents adaptive caching schemes applicable to single and multiple processor systems, and it will be useful for all distributed Web, file system, database, and content delivery services. Among the latest work, Rodriguez et al. (2021) analyse the relationship between recency and frequency, and use ML to achieve optimal scheduling of cache replacement policies. However, these methods use simple heuristics and ML methods, so it is difficult to consider hidden relationship between the cache replacement policies and the workloads, which leads to unsatisfactory performance on complex workloads.

In this paper, we explore the utility of deep reinforcement learning (DRL) in cache replacement policies. Most of the previous research focuses on the characteristics of each individual block in the cache and uses heuristic or ML methods to design cache replacement policies. In contrast, our work is the first to propose cache replacement by learning the relationship between the workloads distribution and cache replacement policies distribution (include LRU and LFU), which allows us to directly train a replacement pol-

icy end-to-end over much more expressive policy features, thus we name this policy *Catcher*[1]. This work focuses on single-level cache replacement and does not consider other mechanisms such as cache prefetching or admission policy.

In summary, our work has the following contributions:

- We propose a general end-to-end cache replacement policy, which uses DRL to model the distribution of requests, for exploring the relationship between the different cache replacement policies and data distribution, which is then proved to be quite useful on cache replacement problem. To our knowledge, this is the first work to study the relationship between the request distribution and cache replacement policy by DRL.
- We design an effective reward function in reinforcement learning (RL), which enables an end-to-end policy optimization system, accelerates the convergence speed of our model, and improves the effectiveness of cache replacement. In addition, we redefine the state in *Catcher*, so that the model can fully learn the distribution characteristics under multi-user conditions, and at the same time ensure that the states in the experience buffer of DRL are independent and identically distributed (i.i.d.).
- By extensive evaluation, we show that *Catcher* outperforms prior state-of-the-art cache replacement policies over a wide variety of workloads in a range of cache sizes. More importantly, *Catcher* would be universally applicable and can be generalized for different cache scenarios.

Related Work

Researchers have proposed ML-based methods for various fields (Ali, Sulaiman, and Ahmad 2014; Zhou and Xiao 2019; Zhang et al. 2019). Multiple recent work apply ML techniques to the cache replacement problem. Ali, Sulaiman, and Ahmad (2014) used well-understood and mature models such as support vector machine (SVM), naïve Bayes classifier (NB), and decision tree (DT) to predict the next visit time of the blocks, which determines the location of these blocks in LRU queue. Jain and Lin (2016) phrased cache replacement as a binary classification problem, where the goal is to predict whether an incoming request is cache-friendly or cache-averse. Similar work is Shi et al. (2019), which used a neural network to predict which blocks are suitable to be kept in the cache. Teran, Wang, and Jiménez (2016) applied perceptron-learning-based reuse prediction to a replacement and bypass optimization, and it shows that cache management based on perceptron learning more than doubles cache efficiency over LRU policy. In addition, some work has been focused on Web caching applications, such as Song et al. (2020), which built three different types of features for each block in the Content Distribution Networks (CDNs), and used a gradient boosting machine (GBM) to improve the cache hit rate. Rodriguez et al. (2021) dynamically determined which replacement policy is used to evict a page by learning the patterns from workloads whenever the eviction operation is triggered. Moreover, Sethumurugan, Yin, and Sartori (2021) proposed a cost-effective cache replacement policy that learns a last-level cache policy with hardware implementation.

In recent years, the RL framework has successfully demonstrated to solve complex problems. Researchers have proposed RL-based algorithms for various system performance optimization tasks like cloud database tuning (Zhang et al. 2019), networks-on-chips (NoC) arbitration (Yin et al. 2020) and hardware prefetching (Bera et al. 2021).

Even though these ML techniques show encouraging results than some heuristics in accurately predicting evicted blocks. However, most memory systems such as processor-level or block-level systems have limited information beyond access addresses. In this paper, we use only the address information of block access to achieve the cache replacement process. In addition, we pose cache replacement as a Markov Decision Process (MDP) of data distribution (state) and multiple replacement policies (action), which is very different from most previous studies that only focused on the features of each block in the cache.

Background and Motivation

Recency and Frequency

There are a few important factors (characteristics) of blocks in the cache that can affect the replacement process including recency, frequency, and size, etc. These factors can be incorporated into the replacement decision. Moreover, a thorough analysis of these factors could benefit cache replacement, yet obviously challenging. Among these factors, recency and frequency are the most important and commonly used factors and have become the research hotspots in recent years. The most representative policy of all recency-based policies is LRU, and the corresponding frequency-based representative policy is LFU. Because every factor that affects cache has its pros and cons for a particular workload, it is very interesting to combine LRU and LFU (i.e., a probability distribution). Lee et al. (2001) confirm the existence of a spectrum of policies that subsumes LRU and LFU policies. Still, while workloads become even more complex, there is an increasing need for an effective approach to intelligently manage the cache which satisfies the requirements and goals under different scenarios by considering the importance of relevant factors. This motivates us to adopt intelligent policies in solving cache replacement problems, and we summarize why this paper chooses recency and frequency as the factors to study the blocks in the cache as follows:

- The representative policies LRU and LFU corresponding to recency and frequency are relatively simple to implement and have stable performance.
- Recency and frequency are the key factors for blocks in the cache, and a reasonable combination can solve all types of requests (Rodriguez et al. 2021).
- Recency and frequency have good *orthogonality*, which is not available in other factors (Lee et al. 2001).

[1] **Cat**s are very sensitive to the environment. We want to design a ca**che** replacement policy that can perceive changes in data distribution as keenly as cats, and become a catche**r** who discovers the distribution of workloads and different cache policies.

Workload Distribution

In addition to designing a cache replacement policy, some studies also focus on analyzing the characteristics of the workload distribution itself and classifying the original type of request data from the workload based on the arrival time. Note that the workload distribution is the same as the distribution of request data, which is just a different expression in this paper. Park and Park (2017) classify characteristics of blocks by combining frequency and reuse distance and expand the block classification into four classes, including FS, FL IS, and IL (FS means frequently accessed, short reuse distance. Other classes see (Park and Park 2017) in detail.). Li and Gu (2020) characterize the patterns of these workloads on a basis of time-series reuse distance trend and classify these workloads into six patterns like Triangle, Clouds, and so on. Similar work includes Chakraborttii and Litz (2020), Rodriguez et al. (2021), etc. These classification methods further enhance our understanding of workloads and guide us in combining different replacement policies. However, most of the classification is just a regular summary of the workload distribution. With the increasing number of multi-user or multi-process scenarios and the increasing complexity of workloads, it is difficult to summarize the distribution characteristics of workloads by simple classification. Such a challenge motivates the need for a more expressive approach that analyzes the workload distribution and finds their relationship with different replacement policies, thus making the approach adaptive to different scenarios.

Deep Reinforcement Learning

Modern application scenarios such as the cloud make it difficult to find the relationship between the replacement policy and the workload distribution through a simple classification. In addition, it is hard to define a clear rule that indicates which replacement policy would be the best choice for a cache in the face of different workload distributions. Because RL has the ability to adapt to dynamic changes in the workload (environment) and handle the non-trivial consequences of chosen policies (actions), it is a good fit for the problems encountered in this paper. We consider cache replacement as a decision-making problem for choosing different replacement policies given the corresponding workload distribution (Joe and Lau 2020). At the same time, to describe the differences between different workloads, we use a neural network (NN) to represent the diverse workload distributions. Considering the above advantages of NN and RL techniques, we are motivated to apply the recency & frequency factors and workload distribution to learn an automatic cache replacement policy in our framework. We conclude the following motivations for using DRL techniques:

- It is difficult to establish a clever mathematical formula to describe the workload distribution. In contrast, NN seems to have better expressiveness and flexibility.
- The idea is to achieve the cache replacement by analyzing the relationship between the distribution of request data and different policies (LRU and LFU). The goal is to optimize the long-term benefits of the cache (hit rate), so RL is required to learn the decision-making process.
- It is important to note that the feedback about the quality of the decision[2] made at any given time in cache is delayed and not instantaneous. This is very similar to the characteristics of *delayed reward* related to RL.

Design

Architectural Overview

Figure 1 shows the workflow of our work, *Catcher*, consisting of three major parts: **Collector**, **Replacer**, and **Learner**. The offline part shows the **Learner** component which is responsible for training a DRL model with different workload distributions. The online part shows the **Collector** component which generates the training data for replay buffer and the **Replacer** component which makes a replacement decision based on the probability distribution of policies generated by the DRL model. In addition, there are some functions, including the reward function, action function, and feature function, which will be introduced later.

The collector mainly provides the raw training data for the DRL model through two state windows (*SW*) and one action window (*AW*). The two *SW* collect the state of adjacent periods, and the state is the access address at each moment. *SW*s obtain the request address vectors $\vec{s}_t$ and $\vec{s}_{t-1}$ in chronological order, where $\vec{s}_t$ and $\vec{s}_{t-1}$ are adjacent but do not overlap in time. Meanwhile, *AW* collects the replacement decisions from $\vec{s}_{t-1}$ to $\vec{s}_t$, including the probability of choosing LRU (a_1) or LFU (a_2) policy in case of cache misses and cache hits (the replacement policy is not selected when the cache hits (**-**), but the probability is set to 0.5). The main part of the learner is the deep deterministic policy gradient (DDPG) (Lillicrap et al. 2019), which is a policy-based RL method with continuous input and output. In addition to training the network, the actor-network in DDPG also needs to output the probability distribution of the replacement policies based on the state vector $\vec{s}_t$ when the cache misses. After receiving the output of the actor-network, the replacer selects either LRU or LFU policy to achieve the cache replacement according to the probability distribution of the replacement policies and updates the information recorded by LRU or LFU policy when the cache hits.

Analysis of Workload Distribution

It is well known that effective cache management requires a good understanding of I/O workload characteristics. The analysis of workload distribution is helpful for NN to gather comprehensive workload characteristics rather than individual requests. LSTM (Long Short-Term Memory) is widely used in previous work and also in *Catcher*, which is designed to learn long, complex patterns within an address sequence, such as reuse distance. However, most existing approaches do not consider the complexity of the workload in the multi-user scenario. Although we can obtain the PIDs of different users or processes, and establish a corresponding analysis model for each user's (or processe) request, this undoubtedly increases the burden on the system. In addition, this

[2]During cache misses, a cache replacement policy inputs the currently accessed block and the cache blocks and outputs which of the blocks in the cache to evict.

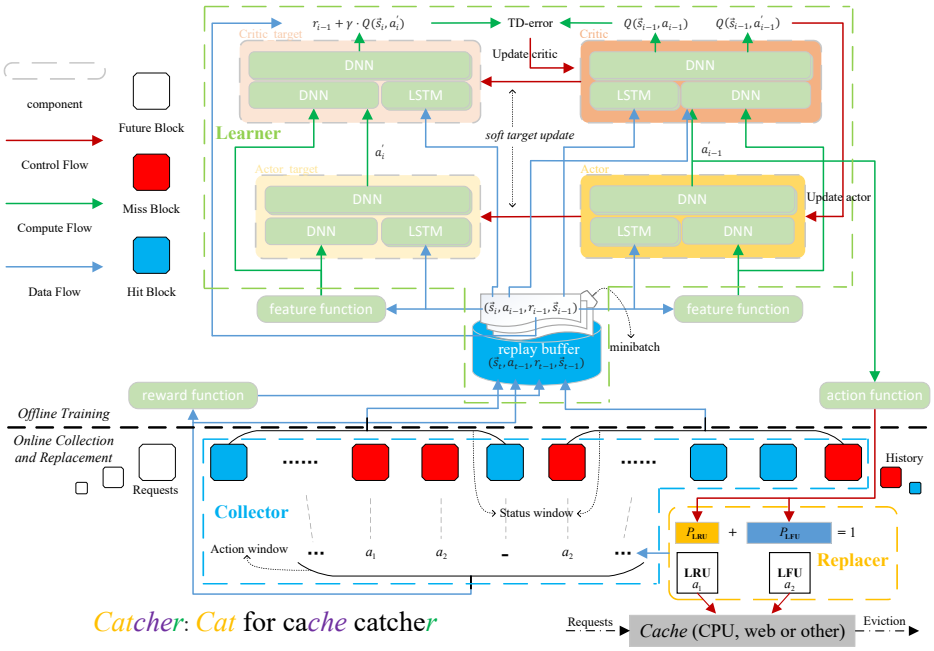

Figure 1: Overview of *Catcher*.

approach will cause more problems in a short-term multi-concurrent user scenario. *Catcher* uses novel time-series methods to select workload features customized, which ensures that *Catcher* can distinguish the number of interleaved workloads in a shared storage system.

To make the sequence more stable, *Catcher* performs the *difference of first order* on the raw address sequence. We use *tsfresh*[3], a third-party package using Python, to rapidly extract a large number of high-information features automatically. Based on the analysis of the importance of up to 1576 features by *CENSUS*[4], we select 10 features as further analyses of complex workloads, which have been proved to be very effective in calculating the number of distinct workloads in a multi-workload setting. This includes *change_quantiles* (includes *mean*, *variance*, and *standard deviation*), *absolute_sum_of_changes*, *fft_coefficient*, *lempel_ziv_complexity*, *count_above_mean*, *count_below_mean*, *longest_strike_above_mean*, and *sum_of_reoccurring_values*, and the description of these features can be found in Christ, Kempa-Liehr, and Feindt 2017. Having an initial address series feature facilitates the better representation of the input address sequence and helps the *shared Dense layers* to learn complex distributions effectively from workloads. Note that our work is the first to propose an analysis of workload distribution based on the address series features of replacement policy and has shown promising prediction outcomes.

DDPG for Catcher

DDPG is the combination of DQN (Deep Q Network) and actor-critic algorithm, and can directly learn the policy. We

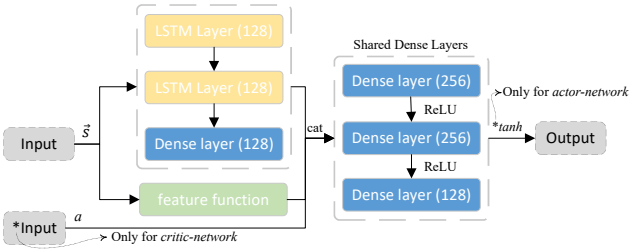

Figure 2: Neural architecture of *Catcher*.

overview the basic neural architecture of DDPG in *Catcher* as shown in Figure 2, which includes actor-network and critic-network. The difference between the critic-network and actor-network is that the input of the critic-network contains action and state, while the output of the actor-network uses the activation function *tanh* to bound in [-1, 1]. We formally define the three pillars of our RL-based *Catcher*: the state vector, the action, and the reward function.

State Vector. The state vector $\vec{s}$ obtained from the collector is processed into two parts (Figure 2), one is sent directly to the LSTM sub-module, and the other is calculated from the feature function to obtain the features of the address sequence. The outputs of the two parts are concatenated and then fed to the *shared Dense layers*. In addition, the distribution of states is not affected by *Catcher* which does conform to the i.i.d. hypothesis between samples in the replay buffer, because the states are independent of our model during the process of continuous generation.

[3]https://tsfresh.readthedocs.io/en/latest/index.html
[4]https://www.cs.emory.edu/ sche422/Census.pdf

Algorithm 1: *Catcher* training algorithm

Randomly initialize critic-network $Q(s,a|\theta^Q)$ and actor-network $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ in DDPG
Initialize target network $Q^{'}$ and $\mu^{'}$ with weights $\theta^{Q^{'}} \leftarrow \theta^Q$, $\theta^{\mu^{'}} \leftarrow \theta^\mu$
Initialize replay buffer R
for step=0 **to** K **do**
  Collect states $\vec{s}_{t-1}$ and $\vec{s}_t$ from the state windows, collect action a_{t-1} from the action window
  Calculate reward r_{t-1} from the *reward function* based on a_{t-1}
  Store transition $(\vec{s}_t, a_{t-1}, r_{t-1}, \vec{s}_{t-1})$ in R
  if step $\equiv 0$ (mod 100) **then**
    Sample a minibatch of N transitions $(\vec{s}_i, a_{i-1}, r_{i-1}, \vec{s}_{i-1})$ from R
    Set $y_{i-1} = r_{i-1} + \gamma Q^{'}(\vec{s}_i, \mu^{'}(\vec{s}_i|\theta^{\mu^{'}})|\theta^{Q^{'}})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_{i-1}\left(y_{i-1} - Q(\vec{s}_{i-1}, a_{i-1}|\theta^Q)\right)^2$
    Update actor by policy gradient: $\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_{i-1} \nabla_a Q(s,a|\theta^Q)|_{s=\vec{s}_{i-1}, a=\mu(\vec{s}_{i-1})} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{\vec{s}_{i-1}}$
    Update the target networks: $\theta^{Q^{'}} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q^{'}}$, $\theta^{\mu^{'}} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu^{'}}$
  end if
end for

Action. In DDPG, the action a in the replay buffer is the ratio of LRU policy ($P_{\mathbf{LRU}}$) selected in *AW*, excluding the number of cache hits (since $P_{\mathbf{LRU}} + P_{\mathbf{LFU}} = 1$ is satisfied, the ratio of LRU policy also reflects the ratio of LFU policy ($P_{\mathbf{LFU}}$)). If there is no cache miss within *AW* at this time, the action is set to 0.5. In order to make the policy probability of the actor-network output $P_{\mathbf{LRU}\sim\mathbf{LFU}}$ satisfy $P_{\mathbf{LRU}} + P_{\mathbf{LFU}} = 1$, we standardize and normalize the output of the actor-network, that is $P_{\mathbf{LRU}\sim\mathbf{LFU}} = (output_{Actor} + 1)/2$ where $output_{Actor}$ represents the output of the actor-network. The replacer in *Catcher* then uses $P_{\mathbf{LRU}\sim\mathbf{LFU}}$ to decide which replacement policy to choose when the cache misses, which avoids the need to predict every block in the cache (replacement policy will determine which block is replaced, *Catcher* does not need to specify exactly). Compared with previous studies (Song et al. 2020; Liu et al. 2020; Shi et al. 2019; Li and Gu 2020), *Catcher* can reduce large-scale operations and improve operational efficiency (when there are many blocks in the cache, the computation overhead and time delay of predicting all blocks will be huge when each round of requests arrives). The actions collected by *AW* in *Catcher* come from the second half of $\vec{s}_t$ and the first half of $\vec{s}_{t-1}$ to reflect the probability distribution of the replacement policy when the state changes from $\vec{s}_{t-1}$ to $\vec{s}_t$. Therefore, the length of *AW* is consistent with *SW*.

Reward. The reward steers the agent towards learning a more optimal replacement policy, so the reward function must be chosen carefully. A simple method is to use cache hit (+1) and cache miss (-1) as a reward at the current time. However, this method is not appropriate for *Catcher* because *Catcher* considers changes in state and action over time, so *Catcher* can set the cache hit rate in *AW* as a reward over time. But the hit rate is always a non-negative value, so *Catcher* cannot get a negative reward. We use independent LRU and LFU as the baseline replacement policies to ensure that *Catcher* can compare with them and generate a negative reward. However, simply considering the performance within *AW* does not guarantee that the overall hit rate of *Catcher* is better than other replacement policies. Based on the above idea, we model the reward function of *Catcher*, which not only considers the difference of performance with the baseline replacement policies at the current time period but also the whole time (Zhang et al. 2019). Formally, let r and $hit_{\vec{s}_1\to\vec{s}_2}$ denote reward and hit rate from $\vec{s}_1$ to $\vec{s}_2$. At time t, we calculate the difference of hit rate Δ from $\vec{s}_{t-1}$ and the initial $\vec{s}_0$ to $\vec{s}_t$ respectively. We design the reward function below:

$$r = \begin{cases} ((1+\Delta hit_{\vec{s}_0\to\vec{s}_t})^\alpha - 1)\cdot(|\,1+\Delta hit_{\vec{s}_{t-1}\to\vec{s}_t}\,|)^\beta & if \;\; \Delta hit_{\vec{s}_0\to\vec{s}_t} > 0 \\ -((1-\Delta hit_{\vec{s}_0\to\vec{s}_t})^\alpha - 1)\cdot(|\,1-\Delta hit_{\vec{s}_{t-1}\to\vec{s}_t}\,|)^\beta & if \;\; \Delta hit_{\vec{s}_0\to\vec{s}_t} \le 0 \end{cases} \tag{1}$$

$$\Delta hit = \begin{cases} \Delta hit_{\vec{s}_0\to\vec{s}_t} = \frac{hit_{\vec{s}_0\to\vec{s}_t}(\boldsymbol{Catcher}) - hit_{\vec{s}_0\to\vec{s}_t}(\boldsymbol{baseline})}{hit_{\vec{s}_0\to\vec{s}_t}(\boldsymbol{baseline})} \\ \Delta hit_{\vec{s}_{t-1}\to\vec{s}_t} = \frac{hit_{\vec{s}_{t-1}\to\vec{s}_t}(\boldsymbol{Catcher}) - hit_{\vec{s}_{t-1}\to\vec{s}_t}(\boldsymbol{baseline})}{hit_{\vec{s}_{t-1}\to\vec{s}_t}(\boldsymbol{baseline})} \end{cases} \tag{2}$$

$$if \;\; hit_{\vec{s}_1\to\vec{s}_2}(\boldsymbol{baseline}) = 0, \; then \;\; \Delta hit = hit_{\vec{s}_1\to\vec{s}_2}(\boldsymbol{Catcher})$$

$$\boldsymbol{baseline} = \begin{cases} \mathbf{LRU} & if \;\; hit_{\vec{s}_0\to\vec{s}_t}(\mathbf{LRU}) > hit_{\vec{s}_0\to\vec{s}_t}(\mathbf{LFU}) \; or \; hit_{\vec{s}_{t-1}\to\vec{s}_t}(\mathbf{LRU}) > hit_{\vec{s}_{t-1}\to\vec{s}_t}(\mathbf{LFU}) \\ \mathbf{LFU} & if \;\; hit_{\vec{s}_0\to\vec{s}_t}(\mathbf{LRU}) \le hit_{\vec{s}_0\to\vec{s}_t}(\mathbf{LFU}) \; or \; hit_{\vec{s}_{t-1}\to\vec{s}_t}(\mathbf{LRU}) \le hit_{\vec{s}_{t-1}\to\vec{s}_t}(\mathbf{LFU}) \end{cases} \tag{3}$$

Algorithm 2: cache replacement algorithm

```
1: for each request block do
2:   if cache is hit then
3:     Update the parameters related to LRU and LFU in
       the cache
4:   else
5:     Collect state vector s⃗ from the state window
6:     Calculate the output a′ of actor based on s⃗
7:     Calculate the possibility P_{LRU∼LFU} from the ac-
       tion function based on a′
8:     Generate a random real number rand from 0 to 1
9:     if rand ∈ [0, P_LRU) then
10:      Cache is managed by LRU policy
11:    else if rand ∈ [P_LRU, 1] then
12:      Cache is managed by LFU policy
13:    end if
14:  end if
15:  Update state windows and action window
16: end for
```

where hyperparameter α controls the impact of the overall hit rate on r (0 to t), while hyperparameter β controls the impact of hit rate on r over the current time ($t-1$ to t), α, $\beta \in \mathbb{N}$. Considering that the ultimate goal of *Catcher* is to achieve better overall performance, we usually set $\alpha > \beta$.

Algorithm 1 shows *Catcher*'s RL-based training algorithm (Lillicrap et al. 2019), which is based on DDPG. Initially, all networks will be initialized randomly and start training when the request arrives (Algorithm 1, lines 1-3). To avoid frequent training caused by concentrated requests in a short time, we plan to train and update the network every 100 requests (Algorithm 1, lines 8-14). Moreover, we use *soft* target updates rather than directly copying the weights when updating the target networks, which greatly improves the stability of learning (Algorithm 1, line 13).

Replacement Decision

Algorithm 2 is the corresponding cache replacement process for cache hits and cache misses. The output of the action function in *Catcher* is the probability of selecting an LRU or LFU policy (Algorithm 2, line 7). When the cache misses, *Catcher* completes the selection of the replacement policy by generating a random number and combining the probability of the replacement policy (Algorithm 2, lines 8-13). Due to the randomness of probability, *Catcher* still has a certain probability of choosing a replacement policy with a small probability, which also makes full use of the exploration & exploitation in RL. Note that each block in the cache needs to record data structure information with LRU and LFU. In addition, the computational overhead of *Catcher* is bound by the computational overhead of LRU or LFU when using LRU and LFU as base policies for participating in cache replacement because it does not have a loop operation in Algorithm 2. In the future, we will improve and combine the common characteristics of LRU and LFU data structures, thereby greatly reducing the time and space overhead of the online component.

Evaluation

Experimental Settings

Workloads. We conduct simulation-based evaluations of several state-of-the-art heuristic and ML algorithms from the caching literature using publicly available production storage I/O workloads. The workloads used in the experiment come from the FIU and MSR datasets in the real environment[5] including 8 production storage traces sourced from 8 different production collections. Each workload has a 1-day duration and contains billions of requests. The amount of original data is specifically large, closing to the scale of Terabyte magnitude. So we use a sampling method to reduce the amount of data. These workloads are used by a large body of prior work, which ensures that we can evaluate the effectiveness of the proposed scheme in general cases.

Configurations. To compare the relative performance of various caching policies, we choose caches that are sized relative to the size of each workload. So cache sizes here will not exceed 1% of the workload used including 0.05%, 0.1%, and 0.5%. The sizes of *SW* and *AW* are consistent with the cache sizes. For all the experiments, we train our model using *Adam* optimizer with a learning rate of 0.001 for actor and critic network with the soft update rate τ = 0.02, and a discounting rate γ = 0.9. The replay buffer R is a finite sized cache (10000) and the actor and critic are updated by sampling a minibatch N = 128 uniformly from the replay buffer. We perform a *grid search* to find the hyperparameters combination that set α as 5 and β as 3.

Baselines. We compare *Catcher* against 9 previously proposed cache replacement policies, including LRU, LFU, ARC (Megiddo and Modha 2003), LIRS (Jiang and Zhang 2002), DLIRS (Li 2018), LRFU (Lee et al. 2001), ACME (Ari et al. 2002), FRD (Park and Park 2017), and CACHEUS (Rodriguez et al. 2021). Both ARC and LIRS are state-of-the-art adaptive policies, and DLIRS is an important extension method of LIRS. CACHEUS is a state-of-the-art ML-based replacement policy, which makes use of recency and frequency characteristics like ARC and LRFU. To make the results more intuitive, we also test the Belady's optimal solution (OPT) (Belady 1966), replaces the block that has the farthest reuse distance among blocks in a cache. OPT is an optimal offline cache policy that is not feasible as online cache. However, it is useful for comparing the maximum performance with that of various cache replacement policies.

In addition, there is previous work on CPU caches (Shi et al. 2019; Sethumurugan, Yin, and Sartori 2021), but these methods mostly focus on hardware caches and also rely on PC or other application features as one of the inputs, which does not exist in the general cache replacement scenarios, their applicability is limited. *Catcher* is implemented using PyTorch[6] and a generic cache simulator[7] including many available cache replacement policies. The simulator has been used in a lot of prior work. To alleviate performance instability caused by RL, we run all experiments

[5] http://iotta.snia.org/traces/block-io

[6] https://pytorch.org

[7] https://github.com/sylab/cacheus

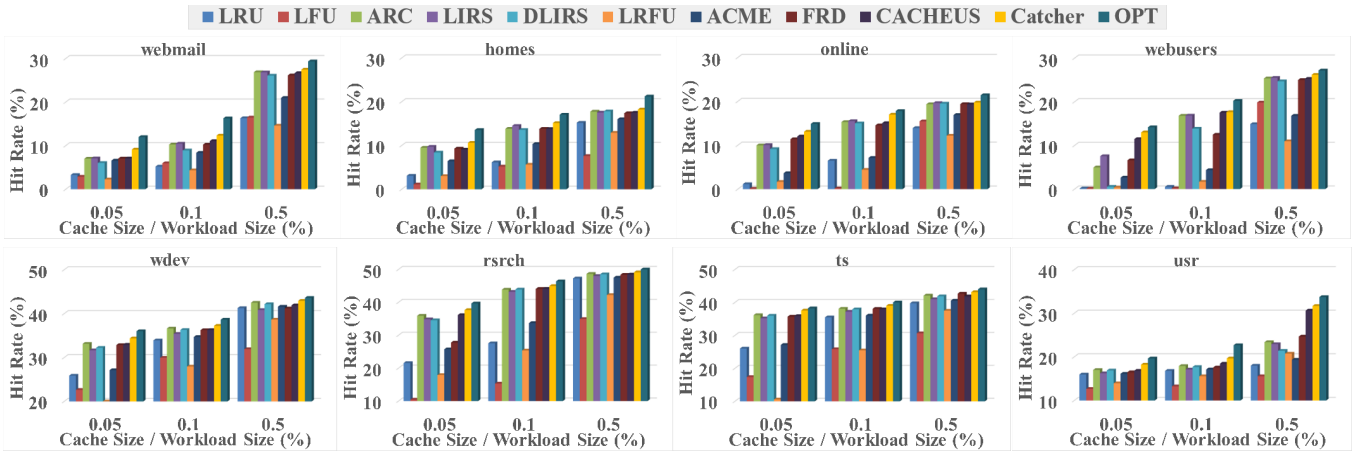

Figure 3: Performance comparison for different cache replacement policies (OPT is the theoretical optimal).

ten times to average as a final result. All the experiments are run on a local Inspur server equipped with a six-core 2.10GHz Intel(R) Xeon(R) E5-2620, 64GiB RAM, and an SMC 512GB hard disk.

Performance Overview

Figure 3 compares the cache hit rate of *Catcher* with different replacement policies, in which OPT is the theoretical optimal. *Catcher* achieves significantly higher cache hit rates than other policies on every workload, ranging from 3% to 150%. Averaged over all workloads, *Catcher* achieves 32%, 22.1%, and 11.3% higher cache hit rates than ARC, LIRS, and CACHEUS when the cache size is 0.05%. Large caches do not benefit from strong replacement policies since working sets are already in cache. Therefore, when cache sizes are large, the difference between the policies cannot be reflected (only 3% to 9% increase when the cache size is 0.5%). When cache sizes are small, *Catcher* improves more because it can make full use of the limited cache space, so subtleties of the replacement policies are observable. Furthermore, the hit rate of *Catcher* is only about 2% to 20% lower than that of OPT, which is the closest method to the optimal value among the currently compared policies.

Further analysis shows that some ML-based cache replacement policies are not better than heuristic algorithms, such as ACME. In addition to the complexity of the workloads, an important reason is that multiple block-related factors are mixed together, which affects the realization of the best performance for each base replacement policy. In contrast, although *Catcher* also combines different replacement policies, it is not an ensemble learning method because only a single cache policy is used to complete the replacement (different cache replacement policies are only for different requests). LIRS also performs better than ARC in some workloads, because most workloads have more requests of the blocks are accessed exactly once. Therefore, LIRS can use 1% of the allocated cache space for buffering, reducing the impact of *cache pollution*. Although DLIRS does not always perform better than LIRS, DLIRS is usually better than LIRS when the performance of LIRS is worse than ARC. This is because DLIRS borrows the idea from ARC and dynamically allocates the cache space to low Inter-Reference Recency (IRR) blocks against high. This experiment demonstrates that *Catcher* can generate a unified and efficient cache replacement policy for generic workloads.

Performance Analysis

To understand the performance advantage of *Catcher* and how *Catcher* differs from LRU, LFU, and OPT, we use a sliding window of the same size as 10 times the cache to record the change in hit rates. In particular, we examine the performance for the webmail (day 16) workload from the FIU trace collection, which is used as a benchmark by many previous works because it contains complete workload types (Park and Park 2017; Rodriguez et al. 2021).

As shown in Figure 4, *Catcher* does not perform very well at first, even worse than LRU or LFU (requests 0∼200). Since the beginning of training, *Catcher* adopts a try-and-error strategy to find and learn the relationship between the probability distribution of replacement policies and workload distribution. It is obvious that *Catcher* gradually adapts to the workload through collecting enough transitions among different states as the number of requests increases, which brings continuous improvement to the performance (requests 300∼700). Finally, compared with OPT, *Catcher* has already achieved a better result in most cases (*Catcher* is close to the theoretical optimal of OPT), indicating that our model owns high efficiency (requests 1100∼2700). We conclude that *Catcher* can adjust quickly from bad replacement decisions and learn how to do better than LRU or LFU. As various types of workloads are collected to the replay buffer, *Catcher* will have better stability and robustness.

Evaluation on Reward Functions

The reward function is vital in RL, which provides impactful feedback between the agent and the environment. For verifying the superiority of different reward functions, the following experiment is designed. We compare *Catcher* with

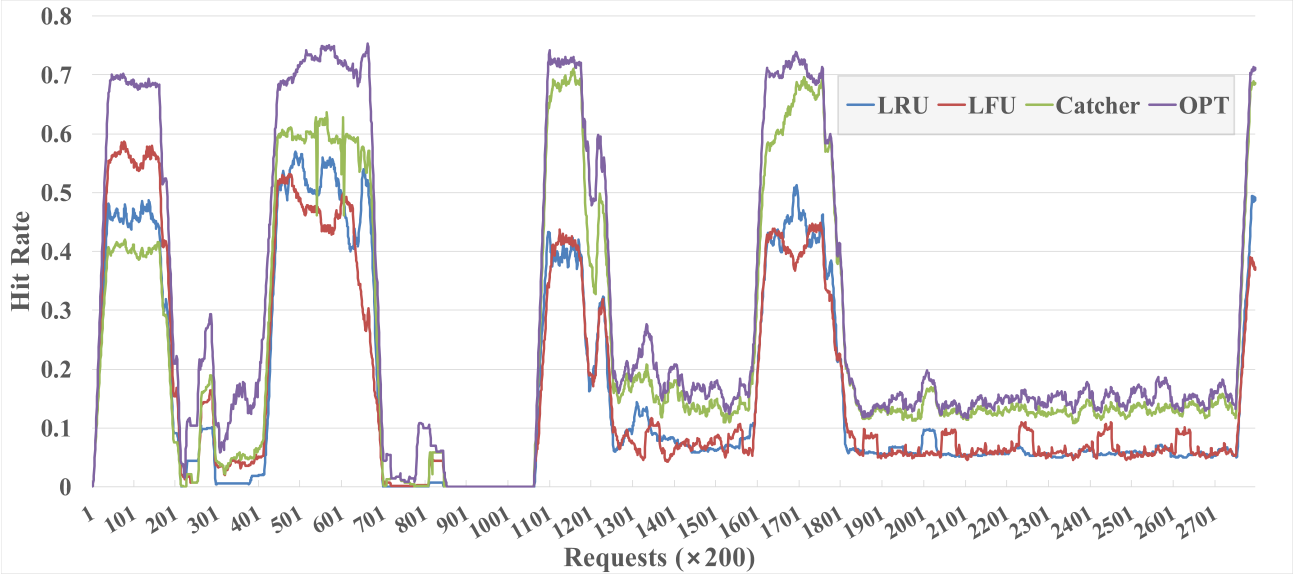

Figure 4: Changes in the hit rates of LRU, LFU, *Catcher*, and OPT on webmail (day 16) workload.

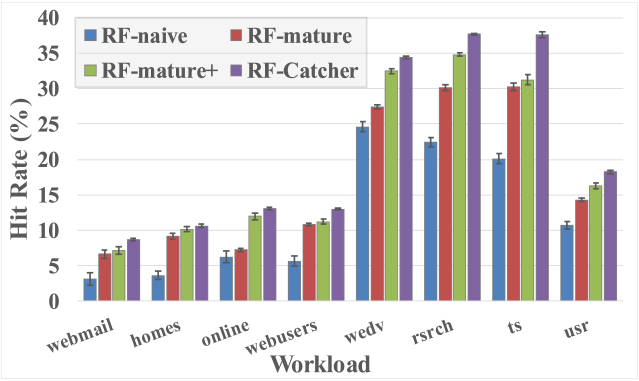

Figure 5: Performance of *Catcher* for all workloads respectively using different reward functions.

other three typical reward functions including

- RF-naive: cache hit r is +1 this moment, otherwise -1.
- RF-mature: take the hit rate of the current period as r.
- RF-mature+: compared with RF-mature, RF-mature+ considers the baseline replacement policies (LRU and LFU) so that r has both positive and negative. The detailed formula is shown as follows:

$$r = \frac{hit_{\vec{s}_{t-1}\to\vec{s}_t}(\boldsymbol{Catcher}) - hit_{\vec{s}_{t-1}\to\vec{s}_t}(\boldsymbol{baseline})}{hit_{\vec{s}_{t-1}\to\vec{s}_t}(\boldsymbol{baseline})}$$

We make a comparison between the three reward functions and our designed RF-*Catcher*. As shown in Figure 5, we adopt 8 different workloads when the cache size is 0.05%. As a whole, RF-naive has the worst performance and is very unstable. In contrast, the performance of RF-mature is better than that of RF-naive. What causes this phenomenon is that RF-naive just considers the performance of the current time, ignoring the hit rate over a period of time. RF-mature+ only achieves a sample target which obtains a better result than the current time period regardless of the whole time performance although it performs better than RF-mature. Especially for workloads with more requests (webmail and ts workloads), if we only consider the performance of the current period, we will gradually lose control of the overall performance and fail to achieve the best performance as the requests continue to come (compared with RF-mature+ on homes and online workloads, RF-*Catcher* only increased hit rates by 4.5% and 9.4%, while RF-*Catcher* on webmail, webusers and ts workloads increased hit rates by 22.5%, 15.9%, and 20.2%). However, RF-mature+ fully demonstrates that a negative r is beneficial to model training and learning. Due to the limitation of the length of the paper, we are unable to provide a detailed discussion and the performance deployed on the reward configurations. In conclusion, compared with others, our proposed RF-*Catcher* takes the above factors into consideration comprehensively and achieves the best performance.

Conclusion

Machine learning is useful in architecture design exploration (Zhang et al. 2019; Zhou, Wang, and Feng 2021; Bera et al. 2021). However, human expertise is still essential in deciphering the ML model, making design trade-offs, and finding practical solutions. In this paper, we propose an end-to-end automatic cache replacement policy *Catcher* that can explore the relationship between the important factors affecting cache replacement and workload. *Catcher* autonomously learns to choose LRU or LFU policy using deep reinforcement learning to achieve cache replacement. Our extensive evaluations show that *Catcher* not only outperforms five state-of-the-art cache replacement policies but also provides robust performance benefits across a wide-range of workloads and cache configurations.

Acknowledgments

This work was supported in part by NSFC No.61832020, No.61821003, No.82090044. Fang Wang and Zhan Shi are the co-corresponding authors.

References

Ali, W.; Sulaiman, S.; and Ahmad, N. 2014. Performance improvement of least-recently-used policy in web proxy cache replacement using supervised machine learning. *International Journal of Advances in Soft Computing & Its Applications*, 6(1).

Ari, I.; Amer, A.; Gramacy, R. B.; Miller, E. L.; Brandt, S. A.; and Long, D. D. 2002. ACME: Adaptive Caching Using Multiple Experts. In *WDAS*, volume 2, 143–158.

Belady, L. A. 1966. A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal*, 5(2): 78–101.

Bera, R.; Kanellopoulos, K.; Nori, A.; Shahroodi, T.; Subramoney, S.; and Mutlu, O. 2021. Pythia: A customizable hardware prefetching framework using online reinforcement learning. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 1121–1137.

Chakraborttii, C.; and Litz, H. 2020. Learning i/o access patterns to improve prefetching in ssds. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, 427–443. Springer.

Christ, M.; Kempa-Liehr, A. W.; and Feindt, M. 2017. Distributed and parallel time series feature extraction for industrial big data applications. arXiv:1610.07717.

Cidon, A.; Eisenman, A.; Alizadeh, M.; and Katti, S. 2016. Cliffhanger: Scaling performance cliffs in web memory caches. In *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*, 379–392.

Jain, A.; and Lin, C. 2016. Back to the future: Leveraging Belady's algorithm for improved cache replacement. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 78–89. IEEE.

Jiang, S.; and Zhang, X. 2002. LIRS: An efficient low interreference recency set replacement policy to improve buffer cache performance. *ACM SIGMETRICS Performance Evaluation Review*, 30(1): 31–42.

Joe, W.; and Lau, H. C. 2020. Deep reinforcement learning approach to solve dynamic vehicle routing problem with stochastic customers. In *Proceedings of the international conference on automated planning and scheduling*, volume 30, 394–402.

Lee, D.; Choi, J.; Kim, J.-H.; Noh, S. H.; Min, S. L.; Cho, Y.; and Kim, C. S. 2001. LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE transactions on Computers*, 50(12): 1352–1361.

Li, C. 2018. DLIRS: Improving low inter-reference recency set cache replacement policy with dynamics. In *Proceedings of the 11th ACM International Systems and Storage Conference*, 59–64.

Li, P.; and Gu, Y. 2020. Learning Forward Reuse Distance. arXiv:2007.15859.

Lillicrap, T. P.; Hunt, J. J.; Pritzel, A.; Heess, N.; Erez, T.; Tassa, Y.; Silver, D.; and Wierstra, D. 2019. Continuous control with deep reinforcement learning. arXiv:1509.02971.

Liu, E.; Hashemi, M.; Swersky, K.; Ranganathan, P.; and Ahn, J. 2020. An imitation learning approach for cache replacement. In *International Conference on Machine Learning*, 6237–6247. PMLR.

Megiddo, N.; and Modha, D. S. 2003. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *Fast*, volume 3, 115–130.

Park, S.; and Park, C. 2017. FRD: A filtering based buffer cache algorithm that considers both frequency and reuse distance. In *Proc. of the 33rd IEEE International Conference on Massive Storage Systems and Technology (MSST)*.

Rodriguez, L. V.; Yusuf, F.; Lyons, S.; Paz, E.; Rangaswami, R.; Liu, J.; Zhao, M.; and Narasimhan, G. 2021. Learning Cache Replacement with {CACHEUS}. In *19th {USENIX} Conference on File and Storage Technologies ({FAST} 21)*, 341–354.

Sethumurugan, S.; Yin, J.; and Sartori, J. 2021. Designing a Cost-Effective Cache Replacement Policy using Machine Learning. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 291–303. IEEE.

Shi, Z.; Huang, X.; Jain, A.; and Lin, C. 2019. Applying deep learning to the cache replacement problem. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 413–425.

Song, Z.; Berger, D. S.; Li, K.; Shaikh, A.; Lloyd, W.; Ghorbani, S.; Kim, C.; Akella, A.; Krishnamurthy, A.; Witchel, E.; et al. 2020. Learning relaxed belady for content distribution network caching. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, 529–544.

Teran, E.; Wang, Z.; and Jiménez, D. A. 2016. Perceptron learning for reuse prediction. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 1–12. IEEE.

Yin, J.; Sethumurugan, S.; Eckert, Y.; Patel, C.; Smith, A.; Morton, E.; Oskin, M.; Jerger, N. E.; and Loh, G. H. 2020. Experiences with ml-driven design: A noc case study. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 637–648. IEEE.

Zhang, J.; Liu, Y.; Zhou, K.; Li, G.; Xiao, Z.; Cheng, B.; Xing, J.; Wang, Y.; Cheng, T.; Liu, L.; et al. 2019. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *Proceedings of the 2019 International Conference on Management of Data*, 415–432.

Zhou, Y.; Wang, F.; and Feng, D. 2021. ASLDP: An Active Semi-supervised Learning method for Disk Failure Prediction. In *50th International Conference on Parallel Processing*, 1–11.

Zhou, Y.; and Xiao, K. 2019. Extracting prerequisite relations among concepts in wikipedia. In *2019 International Joint Conference on Neural Networks (IJCNN)*, 1–8. IEEE.