Optimizing LeCar and Convolutional Neural Network Approaches for Cache Replacement Policy

Yilin Han
yilinhan10@cs.toronto.edu
University of Toronto

Mengxuan Lyu
shawnlyu@cs.toronto.edu
University of Toronto

Abstract

Improving system performance with caching replacement technique is always one of the hottest topics in systematic research. In this project, we proposed a new cache eviction algorithm, namely, LeCar_Opt, which outperforms ARC when the cache size is small. Furthermore, machine learning models are used to learn the cache replacement patterns on real-world workload. We implemented, tuned and analyzed the Multilayer Perceptron, Convolutional Neural Network, and Decision Tree on the OPT solutions. It is concluded that machine learning models are capable of learning OPT patterns with adequate hyperparameter tuning.

1 Introduction

Historically, the development of CPU is much faster than the improvement of disk bandwidth. Cache replacement algorithms as one of the oldest and fundamental computer problems play an essential role to resolve this gap. The performance of caching algorithms strongly influence the system performance. Especially in the past decade, with the advance of the big data technology, more and more data are required to be stored and processed in the modern software systems.

Cache is used to balance the performance difference between high-speed devices and low-speed ones. For example, the data in the memory is written to the Hard Disk Drive (HDD). Since HDD is much slower than the memory, it takes a long time to complete a task in theory, which results in very slow writing performance. In order to resolve the speed difference between memory and HDD, a buffer is created between them to temporarily store the data.

The performance of a cache algorithm is measured by the hit rate. If the hit ratio is low, it means the cache is used inefficiently. Moreover, it may affect the speed of the entire system. In this project, multiple classic and ML-based caching algorithms were investigated. The results were further compared and analyzed with the traditionally caching replacement algorithms.

2 Background

2.1 Existing Cache replacement policies

Traditional caching replacement algorithms mainly utilize the frequency of a page request or the recency of a page request time as a standard to identify which page in the cache needs to be evicted in a replacement operation. The operation in these cache algorithms is relatively simple, and basically, their performance is strongly influenced by the incoming data: they may only perform well on one type of incoming data pattern, and the hit ratio gets significantly reduced on other data requesting patterns. There are two main types of traditional cache algorithms: LRU and LFU.

Least Recently Used (LRU) cache replacement algorithm is the most widely used caching algorithm in the industry. As the name states, The key idea of LRU is performing cache replacement with the least recently used page [9]. The implementation is simple as well. It usually uses a deque [12] and put the most recent page in the tail of the queue. The page is appended to the tail upon a hit on the cache. While replacing a page, it evicts the head page on the queue. This implementation is extremely cheap and the performance is outstanding. In real applications, most of the systems use LRU as their main method. The algorithm shows the superiority to the locality of reference exhibited in the request sequence. However, It is vulnerable to sequential disk scan, such as when the sequential requested data size is larger than the

cache size, LRU has no effect in the cache.

Least Frequently Used (LFU) cache replacement algorithm tends to evict the least frequently used page in the cache. This is based on the assumption that the pages with higher request count are more likely to be requested to be referenced again in the near future [5]. In terms of the implementation, this requires extra overhead to maintain the page number and corresponding frequency counts. Hashmap is normally the best to keep track of the pages. However, the cost to keep the hashmap in the memory could be extremely high when the system is running for a long time. LFU also has disadvantages. It adapts poorly to variate access patterns by accumulating stale pages with past high-frequency counts. For example, some pages are referenced a large number of times at beginning but not used them in the future, these pages would stay in the memory for a long time.

The Adaptive Replacement Cache (ARC) algorithms combine the LRU and LFU algorithms [10]. ARC proposed a novel approach that not only cares about the recency and frequency but also includes the history of evicted pages into considerations [6]. It divides the cache into two LRU lists: T1 and T2. T1 is maintained to cover the short-term utility such that a new page is always stored in the T1 list when it first time entered the cache. The pages in T1 will gradually move to T2 when they are accessed more than once. This means T2 list represents the frequency feature of the page just as LFU. Thus, the long-term utility is preserved in T2. In addition to using T1 and T2 list to represent the cache, ARC also maintains a history list with the same size as cache. They are being called "Ghost" list B1 and B2 were corresponding to T1 and T2 respectively. B1 stores the pages, which is not currently in the cache, that have been recently deleted from T1, and B2 stores the pages recently deleted from T2. B1 and B2 are also acting in LRU manner. The unique characteristic of ARC is the ability of self-tuning. The size of T1 may grow or shrink relative to T2 when there is a hit in B1 or B2. This adaptive feature makes the algorithm capable of adjusting the T1 and T2 size in order to overcome the above-mentioned disadvantages from LRU and LFU.

Belady's Optimal Replacement Policy (OPT) is considered as the best cache replacement algorithm in this field [2] [9]. OPT traces the future blocks that will be brought to cache in order to pick the best candidate to evict. It is not implementable because no one could truly predict the future requests in practice. Nevertheless, it is still used as the performance benchmark for cache algorithms testing.

2.2 Machine learning based approaches

Increasing attention is cast on Machine Learning (ML) in recent years, which is a field where algorithms are applied on computers to learn from previous data for predicting similar patterns in new data, without explicitly programmed. It can be generally categorized into three main approaches: supervised learning, unsupervised learning, and reinforcement learning [11].

Researchers have shown that ML could be applied to improve cache replacement policies: [7] proposed an ML-based caching system to notably decrease the load of content servers by improving caching decisions; [1] developed the ACME system which utilizes ML to learn from multiple cache replacement strategies before making the recommendation about which to evict; [11] showed that supervised learning algorithms like Support Vector Machines and Naive Bayes would lead to considering improving on hit ratio when combined with traditional algorithms like LRU, LFU. Besides, with the increasing computational powers, Neural Networks like MLP and CNN furthermore enhance the learning ability of machines.

As revealing internal relations and capturing hidden patterns among page requests would remarkably improve the efficiencies of cache replacement policies, ML-based approaches showed promising applications in this field. In this work, decision trees [3], MLP, and CNN [13] would be implemented to make decisions over page evictions when there is a cache miss happening.

3 Methodology

3.1 Dataset

The FIU [8] computer science department workload is utilized as the training and testing dataset. The data was collected for three weeks and consist of all I/O traces on two web-servers, one email server, and one file server. Our experiments were conducted on email server data, since it is the most heavily used system in the data set. Each mail data file is roughly 1.8 gigabytes in size. FIU workload trace was generated by blktrace [4] command. Each record from the workload file is corresponding to the nine features of a block request: time-stamp, process id, process name, logical block address, request size, operation type, major device number, minor device number, and block hash. In our experiment, only the integer-typed logical block address is used.

3.2 Implementations

3.2.1 LeCar - Opt

LeCar [14] is a framework that self-proclaims as an Reinforcement Learning (RL) based cache algorithm. It agrees to the fact that a single cache replacement strategy could not satisfy the daily workload scenarios. Both LRU and LFU have drawbacks as described above. With such an assumption, LeCar leverages the reinforcement learning concept, called regret minimization, to switch policies between LRU and LFU. It maintains two weight factors for choosing LRU or LFU policies perspectively and two history list corresponding to the evicted pages after conducting LRU policy or LFU policy. For example. If there is a hit from LRU evicted pages, the weight of using LFU will be increased, and vice versa. With adjusted weights, a random number is generated to select the next policy.

In the early phase of the project, the LeCar algorithm was reproduced, but soon two unacceptable problems were noticed. First, it is not an RL-based algorithm even though they claimed so. LeCar does not have any well-known features that an RL or ML algorithm has and they falsely impose the reinforcement learning concept or terms on the wrong field. For example, reinforcement learning results in a decision matrix during the training and learning process, then the algorithm uses the decision matrix alone in the later testing. However, LeCar is an online algorithm, which makes policy change decision after each weight adjustment. The next policy to be used in purely random and this does not involve any learning or training. The second problem of Lecar is overfitting. The LeCar converges easily on FIU workload with small learning rate and discount rate, thus tuning the algorithm is impossible.

Even through LeCar is problematic, we proposed LeCar_Opt (Algorithm 1) based on the two ideas borrowed from LeCar. First, handling the workload of only two fundamental policies, LRU and LFU. The choice of choosing either policy is dependent on the current environment. In addition, the history of LRU and LFU evicted pages are being tracked.

In Algorithm 1, the LeCar_OPT manages a First In First Out (FIFO)lists of history evicted pages for LRU and LFU perspectively. The size of the list is equivalent to the cache size. The intent is that the algorithm is detecting the patterns from the history list. Specifically, the policy is switched when there is the following cases happens:

- Case 1: If the current policy is LRU and there is a hit on the evicted page history for LRU, besides, if the requested page has the highest frequency among all the pages frequencies in the cache, then the policy is switched to LFU. This is aiming to go against sequential disk read patterns. With LFU, the high-frequency pages will be stored in the cache.

- Case 2: There is also a condition that setup to switch LFU to LRU: if the current policy is LFU, and the hit on the LFU evicted page history list could potentially be a cache hit if the policy was LRU. This is completed by using a counter to record the number of page replacement after the last policy switch. The design is for detecting the recency pattern.

The policy switching is not only decided by whether there is a hit on the corresponding history list but also decided by its recency and frequency as described in the above two cases. At the early stage while developing the algorithm, the hit ratio reduced due to frequent switching of the policy. That is because some pages have stayed in the history pages for too long, causing a false signal to change the policy. For example, the page can stay in LFU evicted page history for a long period of time, that exceed the size. The page is considered invalid because the page is not going to generate a hit even the algorithm uses LRU.

3.2.2 Machine learning based approaches

We borrowed ideas from *Towards a ML based Cache Management Policy* (Maharshi Trivedi, Jay Patel, Shehbaz Jaffer) to preprocess the training and testing data collected from FIU computer science department. To fit the data for the downstream classification tasks, the prepared training data in two ways:

In the first approach, a cache size C is defined, so that the list of logical block addresses could be converted into $cache = \{block_1, block_2, \cdots, block_C\}$. Then, the recency and frequency for each block address are calculated and normalized. Finally, training data: $X = \{X_1, X_2, \cdots, X_C\}, X_i = \{b_i, r_i, f_i\}$ where b_i, r_i, and f_i are referred as the block number, the recency, and the frequency for i^{th} requested block address. Besides, the label is $Y = \{Y_1, Y_2, \cdots, Y_C\}, Y_i \in \{0,1\}$, such that 1 means eviction and 0 means no eviction. The eviction ratio is set in order to reduce the complexity of model training. While retrieving labels, It is assumed that a percentage of OPT blocks are qualified to evict.

In the second approach, a cache size is again defined as C , and the entire cache is store as a single sample such that $X' = \{X'_1, X'_2, \cdots, X'_C\}, X'_i \in \mathbb{R}^{C\times 3}$. In this case the labels are $Y' = \{Y'_1, Y'_2, \cdots, Y'_C\}, Y'_i \in [1, C]$ indicating which page would be evicted.

Let Z be the sample space where samples are represented as $X \times Y$. A model is designed to find a hypothesis

Algorithm 1: LeCar_Opt

Input: requested page q
initialization;
if *q is in cache* **then**
    UPDATE_RECENCY(q);
    UPDATE_FREQUENCY(q);
else
    if *q is in $lru\_evicted\_history$ AND $current\_policy$ is lru* **then**
        $max\_frequency \leftarrow$ MAX(GET_FREQUENCY($cache$));
        if *$FREQUENCY(q) > max\_frequency$* **then**
            $lru\_evicted\_history$.DELETE(q);
    else if *q is in $lfu\_evicted\_history$ AND $current\_policy$ is lfu* **then**
        if *$evicted\_page\_count \geq cache.size$* **then**
            $lfu\_evicted\_history$.DELETE(q));
        end
    end

    if *q is in $lru\_evicted\_history$ AND $current\_policy$ is lru* **then**
        $lru\_evicted\_history$.DELETE(q);
        $current\_policy \leftarrow lfu$;
    else if *q is in $lfu\_evicted\_history$ AND $current\_policy$ is lfu* **then**
        $cache \leftarrow$ SORT(GET_RECENCY($cache$));
        $lfu\_evicted\_history$.DELETE(q);
        $current\_policy \leftarrow lru$;
    end

    if *$cache$ is FULL* **then**
        if *$current\_policy$ is lru* **then**
            $victim \leftarrow$ GET_LEAST_RECENCY($cache$);
            if *$lru\_evicted\_history$ is FULL* **then**
                $lru\_evicted\_history$.DELETE_HEAD();
            end
            $lru\_evicted\_history$.APPEND($victim$);
        else if *$current\_policy$ is lfu* **then**
            $victim \leftarrow$ GET_LEAST_FREQUENCY($cache$);
            if *$lfu\_evicted\_history$ is FULL* **then**
                $lfu\_evicted\_history$.DELETE_HEAD();
            end
            $lfu\_evicted\_history$.APPEND($victim$);
        end
        $cache$.REMOVE($victim$);
    end
    $cache$.ADD(q);
end

space W so that $W \times Z \Rightarrow \mathbb{R}$. To train the model,e the loss is defined as $\iota\left((w,x),y\right), x \in X, y \in Y, w \in W$.

With the data being processed in the first approach, decision trees and MLPs were implemented, which take each sample $X_i \in \mathbb{R}^3$ as the input, and produce $Y_i \in \{0,1\}$ as the output. For the decision tree, Gini Impurity [3] was adopted as the split criterion with equal class weights. During the training, the model would always try to expand until all leaves nodes are pure.

For MLPs, two different structures - MLP1 and MLP2, two models with the highest hit ratio - are implemented to examine the model performances. MLP1 consists of one input layer, one output layer, and three hidden layers. The numbers of neurons for each layer are 3, 10, 100, 500, 2. MLP2 consists of one more hidden layers with a size of 1000. Notably, in both cases, ReLU was applied as the activation function. The model would take each sample as input and compute the class labels, trying to optimize the cross-entropy loss.

$$\text{loss}(x,y) = -\log\left(\frac{\exp(x[y])}{\sum_j \exp(x[j])}\right) \tag{1}$$

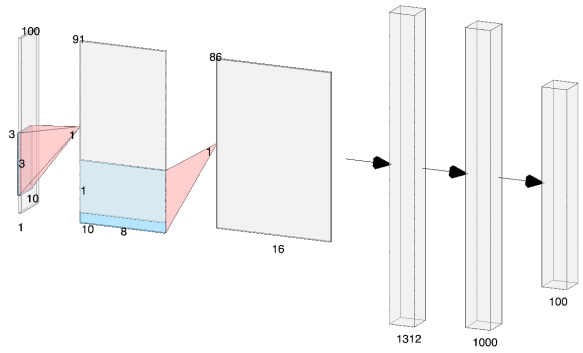

Figure 1: CNN structure

With the data being processed in the second approach, convolutional neural networks were implemented. As shown in the figure 1, the model consists of two convolutional layers and fully connected layers as well. The input layer dimension is (100,3,1) where 100 refers to fixed cache size, 3 refers to 3-dimensional features: block number, recency, and frequency, and 1 is the number of channels. The first convolutional layer consists of 6 kernels with sizes of (10,3,1). Notably, The kernel was set as wide as the data for the model to focus on internal relations among pages in the cache. After two convolutional layers, three layers of fully connected are attached.

4 Experiments and analysis

4.1 LeCar - Opt

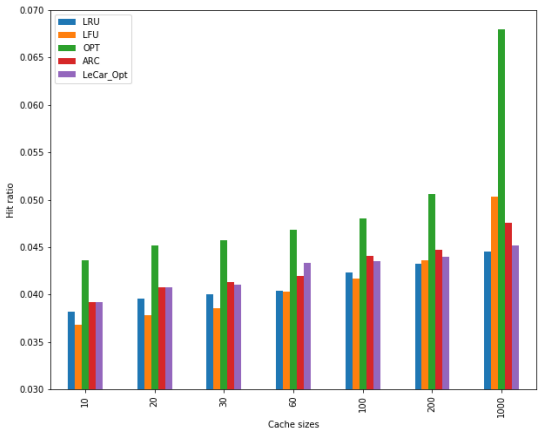

Figure 2: Hit ratio changes with different cache size on FIU workload.

In figure 2, LeCar_Opt is compared with the traditional cache algorithms: LRU, LFU, ARC, and OPT. Among all these algorithms, OPT stands for the best possible hit ratio on the FIU workload. One interesting observation is when the cache size is large, LFU is the best strategy among other algorithms except for OPT, and LRU benefited the least improvement from a larger cache size. This generally show two important characteristics of the workload dataset. First, the pages with high frequency will more likely to be accessed again in the workload. Second, the low performance of LRU with a larger cache size indicates that the workload consists of long and large sequential scan requests. Another interesting observation is that when the cache size is small, LeCar generally outperforms LRU, LFU, and ARC. When the cache size is big, LeCar_Opt is more close to the performance of LRU.

Figure 3 summarizes the efficiency of the algorithm respect to different cache size. The LeCar_Opt is slower than ARC. However, refers to table 1, the execution time increased linearly when the cache size grows.

4.2 Machine learning based approaches

During the experiments with ML-based approaches, hit ratio was used as the evaluation metric. We first tested with MLP with different cache sizes. As shown in the figure 4, a more complex network structure provides little

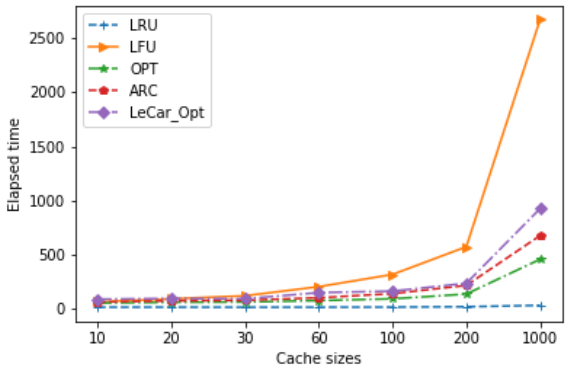

Figure 3: Elapsed time of running each caching algorithm with different cache size.

Cache Size	ARC Elapsed Time	LeCar_OPT Elapsed Time	LeCar_OPT/ARC Ratio
10	64	90	1.40625
20	75	97	1.29333
30	81	95	1.17283
60	104	151	1.45192
100	144	165	1.14583
200	218	237	1.08715
1000	677	926	1.36779

Table 1: Ratio for LeCar_Opt elapsed time and ARC elapsed time respect to different cache size.

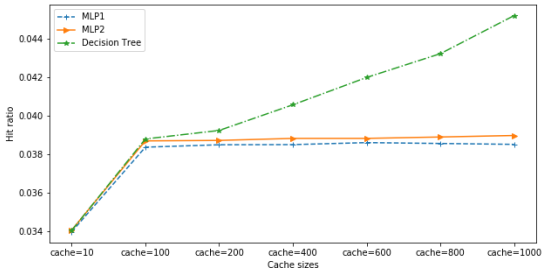

Figure 4: Experiments with MLPs and Decision Tree with different cache sizes.

improvement, and decision tree claimed a better performance compared to MLP. Besides, MLP did not achieve a higher hit ratio in larger cache sizes. On the contrast, the decision tree presented a steadily increased performance with different cache sizes. Then, we can summarize that a decision tree performs much better in the cache replacement scenario.

As shown in figure 5, CNN outperforms both MLP and decision tree, and it is also slightly better than LFU. Our CNN model was constrained to learn internal relations with pages in the cache by fixing the kernel width as the

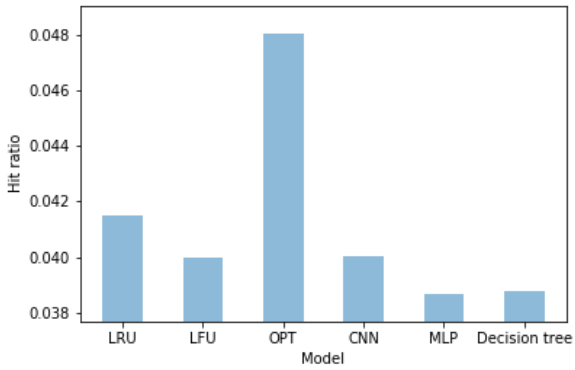

Figure 5: Experiments with different ML-based models

same as page features, and in return, the model captured more hidden features and generated better predictions. In addition, since the designed models are all learning the OPT algorithm, it could be concluded that CNN learns the OPT pattern better than other ML-based models. The result of outperformance is credited to the Kernels [13] design in CNN, which captures the relationships between the inputs.

5 Conclusion and future work

In this project, various traditional or ML-based cache replacement algorithms have been implemented for in-depth investigation. After learning and evaluating the algorithms, one traditional algorithm, LeCar_Opt, was proposed, and three ML-based models were trained for the aim of improving the hit ratio on FIU workload.

The team believes that LeCar_Opt is a promising researching direction on policy switching based algorithms. The experiments result shows that LeCar_Opt achieves excellent hit ratio when the cache size is small, and it is competitive with ARC when the cache size is large. However, there is still room for improvement. The current LeCar_Opt is more sensitive to LRU pattern than LFU pattern. Therefore, it may be a better pattern the algorithm can set up in order to go against long sequential read data.

ML-based models were also implemented for the analysis of cache replacement algorithms. Based on the different features of the models, MLP, CNN and Decision tree were explored and tuned with different hyperparameters. With enough amount of training and high complexity model, it comes to the conclusion that ML-based algorithms are able to learn the OPT patterns on the training dataset.

References

[1] Ismail Ari, Ahmed Amer, Robert B Gramacy, Ethan L Miller, Scott A Brandt, and Darrell DE Long. Acme: Adaptive caching using multiple experts. In *WDAS*, pages 143–158, 2002.

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

[3] Carla E Brodley and Paul E Utgoff. Multivariate decision trees. *Machine learning*, 19(1):45–77, 1995.

[4] Alan D Brunelle. Blktrace user guide. *USA*, 2007.

[5] Edward Grady Coffman and Peter J Denning. *Operating systems theory*, volume 973. prentice-Hall Englewood Cliffs, NJ, 1973.

[6] Mario E Consuegra, Wendy A Martinez, Giri Narasimhan, Raju Rangaswami, Leo Shao, and Giuseppe Vietri. Analyzing adaptive cache replacement strategies. *arXiv preprint arXiv:1503.07624*, 2015.

[7] Youngbin Im, Prasanth Prahladan, Tae Hwan Kim, Yong Geun Hong, and Sangtae Ha. Snn-cache: A practical machine learning-based caching system utilizing the inter-relationships of requests. In *2018 52nd Annual Conference on Information Sciences and Systems (CISS)*, pages 1–6. IEEE, 2018.

[8] Ricardo Koller and Raju Rangaswami. I/o deduplication: Utilizing content similarity to improve i/o performance. *ACM Transactions on Storage (TOS)*, 6(3):13, 2010.

[9] Richard L. Mattson, Jan Gecsei, Donald R. Slutz, and Irving L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems journal*, 9(2):78–117, 1970.

[10] Nimrod Megiddo and Dharmendra S Modha. Arc: A self-tuning, low overhead replacement cache. In *FAST*, volume 3, pages 115–130, 2003.

[11] Muralidharan Murugesan and E Kirubakaran. Impact of machine learning techniques with cache replacement algorithmsinenhancingthe performance of thewebserver. *International Journal of Advanced Research in Computer Science and Software Engineering*, 7:812–816, 06 2017.

[12] Oracle. Deque (java platform se 7), 2018.

[13] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

[14] Giuseppe Vietri, Liana V Rodriguez, Wendy A Martinez, Steven Lyons, Jason Liu, Raju Rangaswami, Ming Zhao, and Giri Narasimhan. Driving cache replacement with ml-based lecar. In *10th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, 2018.