A Machine Learning based Cache Management Policy

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

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

Abstract

Utilizing Caching replacement technique to improve the system performance is always one of the hottest topics in systematic research.

1 Introduction

Abstract:

Utilizing Caching replacement technique to improve the system performance is always one of the hottest topics in systematic research.

Introduction:

Historically, the development of CPU is always faster than the improvement of disk I/O. The gap is getting bigger and bigger. Cache replacement algorithms as one of the core remedy for this difference remains the hotspot in the industry and as well as academy. Especially in the recent ten years, with the advance of the big data technology, more and more data are required to be stored and processed in the modern business software systems. The problem [what is cache, why it is a issue]

our contribution:

Background:

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 need 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 only one type of incoming data pattern, and the hit ratio gets significantly reduced on other data requesting patterns. There are two main type of traditional cache algorithms: LRU and LFU.

LRU(Least Recently Used) 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. The implementation is simple as well. It usually use a deque, and put the most recent page in the tail of the queue. When a hit on the cache, the page is appended to the tail. While replacing a page, it evict the head page on the queue. This implementation is extremely cheap and the performance is outstanding. In the real applications, most of the system use LRU as their main method. The core attribute of the algorithm is it shows the superiority to locality of reference exhibited in 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 cache.

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

The ARC (Adaptive Replacement Cache) algorithms combines the LRU and LFU algorithms. ARC proposed a novel approach that not only cards about the recency and frequency, but also includes the history of evicted pages into considerations. 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 represent the frequency feature of the page just as LFU. Thus, the long-term utility is preserved in T2. In addition to use T1 and T2 list to represent the cache, ARC is also the same number of cache size pages in the history. They are being called “Ghost” list B1 and B2, where 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 as LRU manner. The unique attribute of ARC is it has 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 can adjust 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. 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. Machine learning based approaches.

Increasing attentions are cast on Machine Learning (ML) in recent years, which is a field where people apply algorithms training computers to learn from data, instead of programing them explicitly[Impact of Machine Learning Techniques with Cache Replacement Algorithmsinenhancingthe Performance of theWebserver]. It can be generally categorized into three main approaches: supervised learning, unsupervised learning, and reinforcement learning.

Researchers have shown that ML could be applied to improve cache replacement policies: [SNN-cache: A practical machine learning-based caching system utilizing the inter-relationships of requests] proposed a ML-based caching system to notably decrease the load of content servers by improving caching decisions; [Acme: Adaptive caching using multiple experts] developed the ACME system which utilizes ML to learn from multiple cache replacement strategies before making the recommendation about which to evict; [Impact of Machine Learning Techniques with Cache Replacement Algorithms in enhancing the Performance of the Web server] showed that supervised learning algorithms like Support Vector Machines, and Naive Bayes would lead to considerable improving on HitRatio when combined with traditional algorithms like LRU, LFU. Besides, with the increasing computational powers, Neural Networks like multilayer perceptrons and convolutional neural networks furthermore enhance machines' learning ability.

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, multilayer perceptron, and convolutional neural networks would be implemented to make decisions over page evictions when there is a cache miss happening.

Methodology:

data set [Jens Axboe. blktrace user guide, February 2007.]

We utilize the FIU computer science department workload as the training and tesing data. 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. Since the mail server is heavily used compare to other systems, we mainly ran our experiments on the email server data. Each mail data file is roughly 1.8 gigabytes in size. FIU workload trace was generated by blktrace[] 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, we only operates on the logical block address, where it is an integer type.

Implementation: Classic algorithm approach:

LeCar is a framework that self-claimed as a RL (Reinforcement Learning) based cache algorithm. It agrees the fact that a single cache replacement strategy could not satisfied the daily workload scenarios.Both LRU and LFU has drawbacks as descripted above. With such 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 vise versa. With adjusted weights, a random number is generated to select the next policy.

In the early phase of the project, we were trying to reproduce the LeCar algorithm but soon we noticed that there are two unacceptable problems about it. First, it is not a RL-based algorithm even through 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 a decision matrix during training and learning process, then the algorithm uses the decision matrix alone in the later testing. However, LeCar is an classic online algorithm and making policy change decision with weight adjusting such as a random number is generated after each time weight changes. The next policy is used in purely random. This does not involve any learning or training and the weights is not an applicable model in the future. The second problem of Lecar is Overfitting. When we ran LeCar algorithm on the FIU workload with slightly larger learning rate or discount rate, the algorithm converged, such that it converged the 100

Even through LeCar is not informative, during the experiment, we realize we could still borrow its ideas, and create a classic cache replacement algorithm. We generally believe two ideas are useful to build our own algorithm. First, handling the workload of only two fundatmental pocilies, LRU and LFU. The choice of choosing either policy is dependent on the current environment. The another idea we think its valuable is remain the history lists for LRU and LFU evicted pages. In other words, the history lists are the environment to decide the next ongoing policy.

In Algorithm 1, the LeCar$_OPT manages a FIFO(First In First Out$

$Case 1) if the current policy is LRU and there is a hit on the evicted page h$

$frequency pages will be stored in the cache. Case 2) there is also a condi$

$if the current policy is LFU, and the hit on the LFU evicted page histor$

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 above two cases. At the early stage while developing the algorithm, the policy is switching too frequently that reduce the hit ratio. That is because some pages have stayed in the history pages for too long that providing the false signal to change the policy. For example, the page can stay in LFU evicted page history for really long 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.

Algorithm1 LeCar$_OPT :=$=====================================

$Input : requested page q if q in cache : update_receny(q) update_frequency(q) else : if q is in lru_evicted_history and current_policy is lru : max_frequency := max(frequency_list(cache)) if frequency(q) > max_frequency : lru_evicted_history.delete(q) else if block is in lfu_evicted_history and current_policy is lfu : if evicted_page_count >= cache.size : lfu_evicted_history.delete(q)$

if block is in lru$_evicted_history and current_policy is lru : lru_evicted_history.delete(q) current_policy := lfu else if block is in lfu_evicted_history and current_policy is lfu : cache := sort(get_recency_list(cache)) lfu_evicted_history.delete(q) current_policy := lru$

if cache is full: if current$_policy is lru : victim := get_least_recency(cache) if lru_evicted_history is full : lru_evicted_history.delete_head() lru_evicted_history.append(victim) else if current_policy is lfu : victim := get_least_frequent(cache) if lfu_evicted_history is full : lfu_evicted_history.delete_head() lfu_evicted_history.append(victim)$

cache.remove(victim) update$_recency_frequency(victim) cache.add(q)$ =====================================

Machine learning based approaches

data preprocessing

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, we prepared training data in two ways:

In the first approach, we defined a cache size C, so that we could convert the list of logical block addresses into $cache = \{block_1, block_2, \cdots, block_C\}$. Then, we calculated the recency and the frequency for each block address and normalized them. Finally, we have 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, we also had $Y = \{Y_1, Y_2, \cdots, Y_C\}, Y_i \in \{0, 1\}$ as the label, so that 1 means eviction and 0 means no eviction. When retrieving labels using OPT, we did not evict one page. Instead, we set the eviction ratio so that multiple pages would be labeled as evicted for each cache miss.

In the second approach, we also defined a cache size C, and stored the whole cache as a single sample such that $X' = \{X'_1, X'_2, \cdots, X'_C\}, X'_i \in \mathbb{R}^{C\times3}$. In this case our labels are $Y' = \{Y'_1, Y'_2, \cdots, Y'_C\}, Y'_i \in [1, C]$ indicating which page would be evicted.

ML

Let Z be the sample space where samples are represented as $X \times Y$. Our model is designed to find a hypothesis space W so that $W \times Z \Rightarrow \mathbb{R}$. To train our model, we define the loss as $\iota((w,x),y), x \in X, y \in Y, w \in W$.

With the data being processed in the first approach, we implemented decision trees and multilayer perceptrons 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, we adopted Gini Impurity 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 the multilayer perceptrons, we implemented with different number of layers, different size of each layer to examine the model performances.

With the data being processed in the second approach, we implemented convolutional neural networks. As shown in graph , the model is consisted of two convolutional layers and fully connected layers as well. Notably, We set the kernel as wide as the data to let the model focus on internal relations among pages in the cache. We also tried with different sizes and structures, including the number of convolutional layers, the size of fully connected layers, and size of kernels.

2 Experimental setup

3 Analysis

4 Discussion

References