Towards a ML based Cache Management Policy

Maharshi Trivedi
1004212470
maharshi.trivedi@mail.utoronto.ca

Jay Patel
1003747886
jayk.patel@mail.utoronto.ca

Shehbaz Jaffer
1002273817
shehbaz@cs.toronto.edu

1 Abstract

Enterprise data centers store persistent data in different storage media, varying in speed and capacity. Reducing overall latency incurred in serving requests involves migrating frequently accessed hot data from slower to faster media (fetching) and moving colder data from faster to slower media (eviction). Managing data involves conventional, hard-coded algorithms that either do not reflect the access patterns of enterprise workloads (LRU, LFU, FIFO) or are practically unimplementable (OPT). In this paper, we analyze real-world workload traces and evaluate 3 machine learning based approaches - Logistic Regression, Neural Network and k-NN for block eviction over hybrid media. Second, we propose two novel hyper parameters - sampling_frequency and evict count for cache replacement policy. Third, we architect two data models - Block Cache and Vectorized Cache for block classification. Finally, our proposed block eviction technique is scalable across cache sizes when a parametric Machine Learning algorithm is used.

2 Introduction

Data centers often employ a low capacity, faster media like Non-Volatile RAM in tandem with comparatively slower, large capacity storage media like Hard Drive to increase the overall read and write service request latency. Based on the current process, CPU requests the data which is called *block* in unit granularity to perform the read/write objective. Particular blocks from slower media are selectively stored in faster media, and most read-write requests should be served from faster media. Any request that is served from slower media takes more time and often is the cause for high, unwanted latency. One way to reduce high latency is predicting which blocks should be stored in the limited fast media so that most requests are served directly from the faster media. A number of algorithms have been proposed in order to manage the fast media. These algorithms decide which blocks should be evicted from the faster media to serve newer blocks and therefore known as *Page Replacement Algorithms* or *Cache Replacement Policies*. Usually, these algorithms are hard-coded and do not truly reflect the access pattern of real-world workloads.
In this project, we try to implement various ML techniques to model an ML powered page replacement policy that can efficiently *classify* blocks into *cache blocks* - i.e. blocks that lie in the faster media, and *non-cache blocks*, based on features of the block. We further provide a performance comparison between existing cache replacement techniques and our ML approach for managing data blocks in cache.

32nd Conference on Neural Information Processing Systems (NeurIPS 2018), Montréal, Canada.

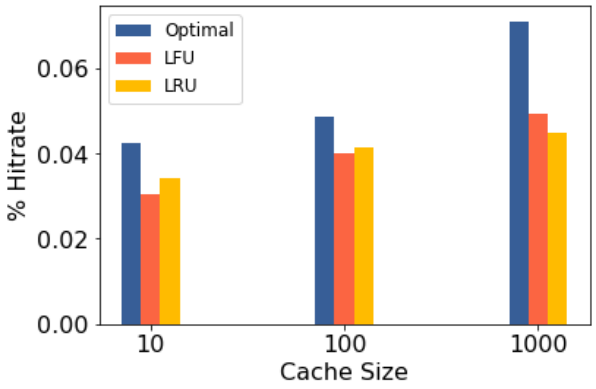


Figure 1: **Evaluation of different cache replacement policies for 10, 100 and 1000 cache sizes with respect to *Hitrate*. The OPT algorithm has the highest Hit Rate, followed by Least Recently Used, Least Frequently Used algorithm.**

3 Background

3.1 Existing Cache Replacement Policies

When a block is requested by an application, the OS first checks the availability of the requested block in the cache. If the block exists in the cache then the request is said to *hit* the cache. If the requested block does not exist in the cache, a cache *miss* is said to occur and the system replaces one of the blocks present in the cache with the requested block. Identifying which block should be evicted from cache to account for the missed block correctly is critical to achieve good workload performance. *Effectiveness of different cache replacement policies can be decided based on the Hitrate. Hitrate is the ratio of number of cache hits to the total number of requests.*

Decision about the eviction of a block in the cache is taken with many parameters recency, frequency, sequence of blocks with which they were added, future sequence of the block with which they will be requested. More detailed and comprehensive explanation about all the cache replacement policies can be found in [**?**]. Some of the contemporary cache replacement policies are enumerated below:

1. **Belady's Optimal Replacement Policy (OPT)** The OPT algorithm is the ideal replacement algorithm that causes the least number of page cache *Miss*. OPT looks at the future sequence of the trace blocks and based on that evicts a block that will be accessed furthest in the future and retains all the other blocks in the cache. This results in very low number of cache *Miss* and higher *Hitrate*. This is not-implementable as future requests or sequence of workload traces is almost impossible to predict.
2. **Least Recently Used (LRU)** LRU keeps the track record of recency property of all the blocks in the cache. When eviction takes place, LRU evicts the block which has been accessed recently. More recently a block has been accessed, more likely that same block will be requested in the near future.
3. **Least Frequently Used (LFU)** LFU keeps track record of how frequently a block in the cache was requested in history. More frequently a block was accessed in past, more likely it will be accessed in the near future. It evicts a block which was accessed least in the past and retains all the other block in cache, when *Miss* occurs.

Evaluation of different cache replacement policies with respect to its *Hitrate* is shown for different cache sizes in Figure 1. As the OPT algorithm gives the highest *Hitrate*, our aim is to train ML (classification) algorithm such it can work like an OPT. Therefore, for all the ML models which will be discussed in the following sections will be supervised and labelled with OPT.

Features	Unit	Used
Time-stamp	Nanoseconds	No
Process ID	int	No
Process Name	str	No
Logical Block Address	int	Yes
Request Size	int	No
Operation Type	Read/Write	No
Major Device Number	int	No
Minor Device Number	int	No
Block Hash	sha256	No
Frequency	int	Yes
Recency	int	Yes

Table 1: **Block Features**

4 Implementation

4.1 Data Set

The FIU workload [**?**] consists of enterprise block traces of users collected over a duration of three weeks. We limit our analysis to workload traces spanning across a time period of 48 hours. Each I/O trace consists of a virtual machine running two web-servers (web-vm workload), an email server (mail workload) and a file server (homes workload). The workload trace for each day is approximately 2 GB in size. Each row of the trace corresponds to one block and its nine features. Table 1 summarizes the different features and its types. data distribution of a typical workload trace along 3 major features is shown in Figure 2.

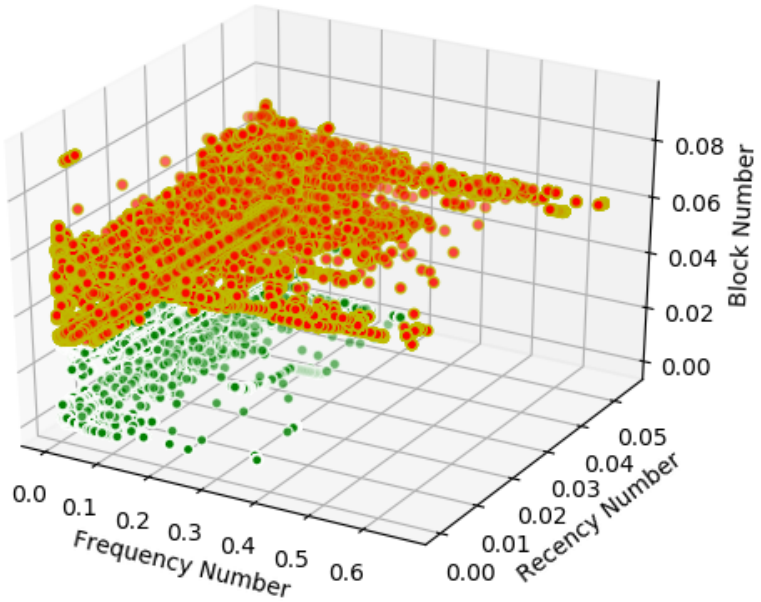


Figure 2: **Visualization of 3-dimensional feature space. A point in the 3-dimensional space represents a cache block. Red points represent evicted blocks from cache, while Green points represent retained blocks in cache. Data is generated by running OPT algorithm on a day long workload trace. Axis of 3D feature spaces are Frequency number, Recency number and Block number in X, Y and Z direction, respectively. From the visualization, the data is separable along Block number axis.**

To deal with this large data set, we use two approaches to train ML model. First, we evaluate the evictions and *Hitrate* for a single day's block-trace. Second, we train the model on the workload of one day and evaluate the evictions and *Hitrate* for the next day.

4.2 Data Gathering

Our objective is to emulate OPT algorithm which gives us the highest possible *Hitrate*. We use OPT algorithm to predict the most desirable block that should be evicted from the cache. For each Cache configuration, we store features of blocks within the cache. For each Cache instance of blocks, we store features of all blocks in the Cache. Further, we generate the blocks that would be evicted by OPT given the current Cache configuration and blocktrace. For labels, the block which is to be evicted from OPT is assigned label 1 and all the other blocks which are retained in the cache is assigned label 0. Listing 1 summarizes our approch for data gathering.

```
for each block in blockTrace:
    if block not in Cache:
        Cache = Cache + block
        x = features(Cache)
        y = OPT(x, blockTrace)
        X = X + x
        Y = Y + y
```

Listing 1: **Data Gathering phase. for each miss operation, we generate a sample x and the corresponding list of evicted blocks y. The sample and corresponding bast eviction candidates are stored in X and Y.**

4.3 Data Modelling

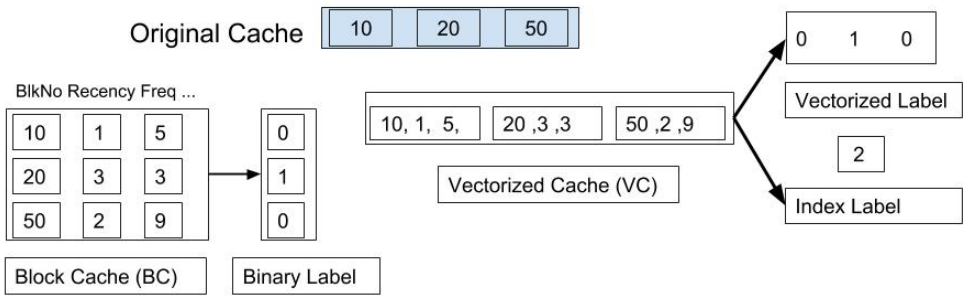


Figure 3: **Data model for cache of size 3 containing block numbers 10,20 and 50. we set eviction blocks to 1. BC representation arranges each block features eg. block number, recency and frequency in a row. Binary labels show 1 for eviction and 0 for non-eviction. Vectorized Cache is a vector of all block features of cache in a linear line. Vectorized label shows a vector of 0's and 1's representing non-evicted and evicted blocks. Index label signifies block number in cache that should be removed.**

We represent both our data and labels in different configurations. Figure 2 summarizes the data and label representations.

4.3.1 Data Representation

1. **Block Cache Representation (BC)**- In this configuration, each row of a sample represents a block in the cache. Each column for a row represents a feature of that particular block in the cache.
2. **Vectorized Cache Representation (VC)**- In this structure, each row of a sample represents all the blocks in the cache. First N columns of a row represents N features of the first block in the cache. Thus, the final size of a row will be `cache_size * number_of_features` of each block.

4.3.2 Label Representation

1. **Binary Label (BL)**- In this configuration, each row of the label corresponds to the one block of the cache. 1 in each row shows that the block is to be evicted and 0 in the row represents that the block is to be retained.
2. **Index Label (IL)**- In this arrangement, instead of 0/1, index of the block which is evicted with OPT will be stored as a single element.
3. **Vectorized Label (VL)**- This is the transpose arrangement of Binary Label. The entire cache line is stored as a vector of 0's (non-evicted blocks) and 1's (eviction blocks).

While implementing different ML algorithms, BC can only be used with BL, However, VC can be used with both IL and VL.

4.3.3 Data Preprocessing

We select recency count, frequency count and block number as the features of all the blocks, present in the cache. Block features are normalized as some ML algorithms are sensitive to higher dimensions.

4.4 Training

We pre-set three hyper-parameters - `cache_size`, `eviction_count` and `sampling_frequency`. We define `eviction_count` as the number of blocks that we select in advance as eviction candidates on a cache miss. `sampling_frequency` is the number of iterations after which we should gather cache block features and labels. After each `sampling_frequency`, we compute the best eviction candidates using OPT algorithm. We train our model using a instance of cache and eviction candidates selected by OPT algorithm. We list the **problems and solutions** training our data set below:

1. To improve **scalability** with increased cache size, we predict multiple eviction blocks simultaneously as a percentage of `cache_size`. For a single eviction candidate, our accuracy decreases considerably with increased cache size. However, with approximating the next `evict_count` blocks, there are multiple overlaps between OPT and our algorithms eviction candidate blocks.
2. To reduce **bias** during training due to large non-evicted blocks as compared to evicted blocks, We only select the top `eviction_count` blocks having best eviction probability instead of all blocks predicted by our classifier. Further, we place the requested block at the same location as the evicted block. This improves our class variance in Vectorized Cache data model.
3. To reduce **over-fitting**, we introduce `sampling_frequency`. This ensures we do not train multiple caches with non-evicted blocks, as the blocks accessed with higher frequency will be labelled as non-evicted blocks multiple times.

4.5 Testing

Each testing phase involves iterating through the block trace and filling the cache. Once the cache is full, for any referenced block not in the cache we call `ML_evict()`. This gives us `eviction_count` blocks that are the best candidates for eviction. Listing 2 summarizes our Testing phase.

```
for each requested_blk in testTrace:
    if requested_blk not in Cache:
        x = features(Cache)
        evict_blk = ML_evict()
        Cache = Cache - evict_blk
        Cache = Cache + requested_blk
```

Listing 2: **Testing Phase. If the requested block is not in the cache then trained ML model trained with OPT is called to predict the block number and cache index at which the block is to be evicted in the cache. Requested block is added at the same place where the block in the current cache is evicted.**

4.6 ML Algorithms

We use a combination of different data models and supervised ML algorithms. First, we describe the different supervised ML algorithms we use to implement Cache Eviction. Next, we compare cache eviction accuracy of ML algorithms and and *Hitrate*:

1. **Logistic Regression**- We first use Logistic Regression to classify eviction blocks and non-eviction blocks. We use both Binary Labels and Index Labels (BC and IL; Section 4.3.2) to train the model. Binary labels (BL) require less data to correctly identify labels as compared to Index labels (IL). This is because BL only generates two labels, however IL requires multiple cache instances to train for each index of the cache. Consequently, large cache sizes require a large number of cache instances. On the other hand, we observe that Index Labelling (IL) provides higher accuracy. Unlike BL, IL uses Vectorized Cache Representation (VC) of the feature space, which helps it form better association between features of blocks within a cache.
2. **Neural Network**- We treat Neural network as a group of logistic classifiers. With Neural Network, we use both Index Label (IL) and Vectorized Label (VL) with Vectorized Cache Representation (VCR) of the feature space.
3. **k- Nearest Neighbor**- kNN does not involve training overhead. We made an attempt to fit both Block Cache Representation (BC) and Vectorized Cache Representation (VC). The major impediment we observed while implementing k-NN with VC representation was the *Curse of Dimensionality*. Vectorized representation of a particular cache has size `= cache_size * number_of_features` of each block, which results in to a very high dimensional space, even with a lower Cache Size. However with Block Cache representation, each blocks eviction or non-eviction is predicted separately. Hence, the feature space is constant in size and we do not incur the curse of dimensionality.

5 Results

5.1 Effect of Cache Size

We are successfully able to scale each of the machine learning algorithms over large cache sizes. This is because of our `evict_count` parameter predicts a percentage of eviction blocks at once instead of calling eviction for each block for every iteration as shown in Figure 4a. We are unable exceed the hit rate of LRU and LFU since we have not exhaustively explored the different hyper parameters for KNN (k), LR (learning rate) and NN (depth and width). Other hyper parameters eg. `evict_count` have also not been explored completely due to time and resource constraints.

5.2 Effect of Test/Train Split

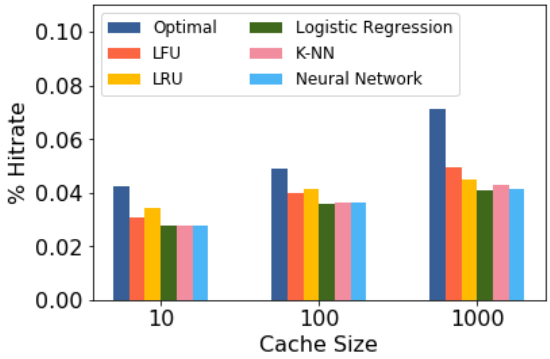
We observe that data modeling has a significant impact on hit rate and accuracy result. For Block Cache model, we observe that the eviction accuracy is more 4d however the hit rate is lesser 4c than Vector Cache representation. This is because we loose inter-block relation within a cache in BC model. Using the Vector Cache representation, the entire cache is used to predict the eviction blocks. Recall that Vector Representation computationally costlier to train due to increased feature space. Hence there is a trade-off between better accuracy and training scalability.

5.3 Effect of eviction count

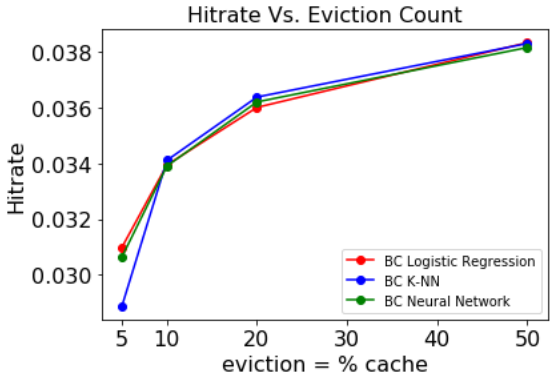
In addition to our ML algorithm, we tune the number of eviction candidate blocks that we request our ML algorithm to predict. We observe that the higher the percentage of eviction candidates, the better is our Hitrate. Figure 4b shows how increasing the eviction count from 5 to 50% increases our Hitrate by 10%. This shows that with increased dependency on ML prediction, our hitrate increases significantly.

6 Related Work

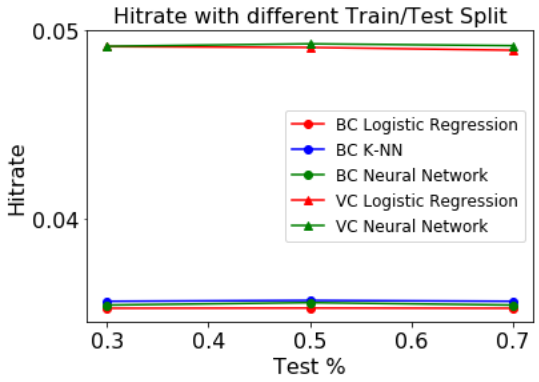
Several researchers applied Machine Learning methods to this caching problem [**? ? ? ?**]. *Robert B et al.* (2002) combined different cache replacement policies (like LRU, LFU and FIFO) and assigned



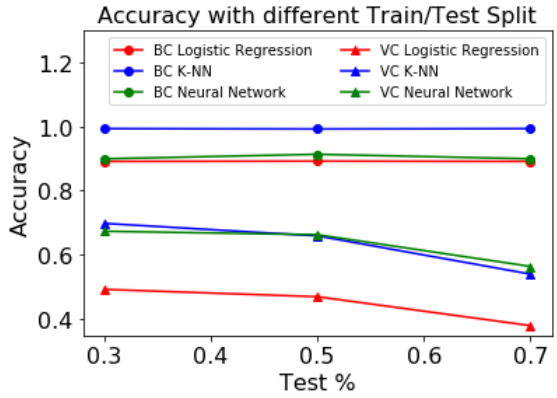
(a) *Effect of Cache Size on Hitrate*



(b) *Effect of Eviction Count on Hitrate*



(c) *Effect of Test Train split on Hitrate*



(d) *Effect of Test Train split on accuracy*

Figure 4: Hitrate and Accuracy plots: (a) Effect of Cache Size on Hitrate for different Cache Replacement policy. OPT has the highest Hit Rate. All 3 ML approaches give better hit rate with increased cache size. k-NN is better than LR and NN (b) Plot of eviction count versus Hitrate. Higher the eviction count, higher is the Hitrate. (c) Effect of Test Train split on hit rate and (d) Effect of Test Train split on accuracy. VC model gives better hit rate than BC model across test/train split.

votes to all of them.[**?**] *Giuseppe V et al.* (2018) used probability distribution of two static cache replacement policies: LRU and LFU.[**? ?**] *Ignacio C et al.* (2017) evaluated the efficiency of the execution policy of Curator (a background MapReduce-style execution framework) with the help of a Machine Learning model.[**?**] Contemporary cache replacement algorithms have been explored using various machine learning techniques for application agnostic workloads[**?**], Online Machine Learning Algorithms [**? ?**] and Deep Learning[**?**]. Previous work [**?**] has found considerable gains in object recognition for images. ARC[**?**] and CAR[**?**] create two caches - L1 and L2, where it places blocks that occur once in L1, and blocks that occur twice in L2. This technique enables them to bypass "scan" based workloads, which replaces all blocks in the LRU cache.
We plan to apply this research in finding relevant blocks for pre-fetching. Finally, we plan to use one or multiple such techniques and demonstrate performance comparison with respect to state of the art caching polices.

7 Limitations and Future Work

1. Each block in our workload trace has 9 primary parameters and 2 derived parameters - frequency and recency count. **Different variations of these 9 features** can be implemented to predict evictions.

2. Our two different data representation (BC and VC) gave ***Hitrate* slightly lower than LRU**. Our attempt was to make a cache replacement policy that can mimic OPT and can give higher *Hitrate* than LRU.
3. Cache replacement policy based on **k-NN** with Vectorized Cache Representation (VC) is computationally very inefficient as our features increase with increased cache size.

We propose the following techniques to improve the *Hitrate* of our cache replacement policy.

1. Essentially, we formulated the problem as the supervised learning problem. However, unsupervised methods such as **K-Means Clustering** can also be applied to this problem. Moreover, We applied all the supervised ML algorithm separately and trained each of them individually.
2. **Synergy of different ML algorithms** with some weights can also be tried to evaluate the effectiveness. In all the supervised ML algorithms, hyper-parameters of each algorithm can also be tuned with the help of **Grid search**.
3. Cache eviction problem can also be thought of an optimized resource allocation problem, **Multi-Armed-Bandit problem** [**?**]. Multi-Armed-Bandit falls under the category of Reinforcement Learning. Various other **Reinforcement Learning** methods can also be tried for cache replacement policy. Sequential requests of the blocks can also be formulated as a time sequence problem. This temporal dynamic behaviour can be tackled with the help of **Recurrent Neural Network (RNN)**.
4. For now we evaluate performance of ML with BC and VC representation of the data and trained ML model on a day long trace only. We can use traces from multiple days/weeks and train the ML model on all the traces instead of just one trace.
5. Finally, cache management involves two techniques - caching and fetching. currently, fetching of blocks is done sequentially without taking relevant block features into consideration. One way to improve cache replacement policies could be to learn which data blocks would be next referenced and fetch those blocks into cache [**?**].

8 Conclusion

In this project, we evaluate cache eviction policy using different ML algorithms. We implement and evaluat the performance of supervised ML algorithms as an option to the contemporary cache replacement policies such as LRU, LFU and OPT. We select 3 parameters: Frequency count, Recency count and Block number for each block in cache. We train 3 different ML models- Logistic Regression, k-NN and Neural Network, with two different cache representations: Block Cache Representation (BC) and Vectorized Cache Representation (VC). We evaluate and compare performance in terms of *Hitrate* and *block eviction accuracy* of each ML based algorithm with existing cache replacement policies (LRU, LFU and OPT). This *Hitrate* comparison of ML algorithm with existing cache replacement policies are evaluated for both BC and VC representation of the cache.