Cache Eviction Policies using ML-based rules

Bhaargav Sriraman, Mandakinee Singh Patel, Suresh Siddharth
{srira048, patel908, siddh010}@umn.edu

1 Abstract

The effectiveness of cache eviction policies is a major indication of cache performance. As cache sizes become smaller with respect to the data they handle, the importance of the performance of these policies are even more important. In this paper, we present a machine learning driven approach to designing a cache eviction policy for a distributed memory object cache system. Such systems are generic in nature, but are mainly geared towards speeding up dynamic web applications. The system is evaluated on synthetically generated data set over parameters like cache size, and model choices. We also discuss certain design choices made, and have metrics on the overall performance of the system.

2 Introduction and Problem Statement

Caching has always been one of the fundamental concepts when it comes to computing. It is widely used in a plethora of applications like web servers, databases, file systems, operating systems, etc. in order to speed up the processing time. However, when compared to primary storage, caches are both more expensive and smaller in size.

Traditionally, most computation would happen at large data centers where - relatively - memory, storage, and computation resources are abundant. However, there is a recent trend of distributed processing, and even beyond that, processing beyond the edge network - on IoT and mobile devices. In such environments, cache sizes are expected to be a lot lower than what you would expect in a machine at a large data center. However, the data-flow through these devices is still expected to remain as varied as before. This leads to problems where not all data can be cached and cache-replacement policies become crucial in maintaining cache performance. Historically heuristic based cache replacement policies are efficient for some workloads but perform poorly in others [2]. In the project, we aim to develop a policy that works well for all workloads. Hence, we propose an ML-based cache replacement model that can possibly outperform traditional cache replacement policies like LFU/LRU, etc.

3 Related Work

The cache replacement problem has optimized in many ways over the years. Megiddo et al. [3] proposed ARC (Adaptive Replacement Cache) which introduced a concept of a cache that evolves which varying access patterns in an online manner.ARC works uniformly well across varied workloads and cache sizes without any need for workload specific a priori knowledge or tuning. However, ARC starts showing performance degradation when working with caches with smaller sizes - when a “stable” working set does not fit in the cache (referred to as “ARChilles’ Heel” [1]).

Another approach was used by ACME by Ari et al. [3], which followed a "system of experts" which proposed different strategies. Each "expert" was given a weight which represented the recent performance of that expert, and decisions were made based on the expert which had the highest weight.

Finally, Vietri et al. [7] thought to improve performance further by building LeCar which uses Machine Learning based cache replacement policies and was able to show that these policies outperformed existing works, especially when working with caches of a limited size. It dynamic balances two orthogonal algorithms by using reinforcement learning to gradually adjust the weight (contribution) of these algorithms to adapt to real-time changing request workloads.

In this project, we aim to build on top of this effort and improve on the ML-based models, aiming for a model that directly predicts the eviction candidate.

4 Implementation

4.1 System Architecture

Inspired by the architecture of the popular open source distributed cache, Memcached, our system consists of a centralized cache controller which receives requests from the various application servers [4]. The controller then identifies the corresponding cache server in which the data resides/should reside, and forwards the request. Cache servers do not replicate data among themselves, and there is no cross-talk. This provides us with a minimum viable distributed cache baseline on which different policies, and enhancements can be evaluated.

The decision to not use an existing open source solution was done after considering the effort involved in understanding the implementation and ability for easy customization on top of the system. By developing a simple distributed cache from scratch, we would also be able to confidently evaluate our ideas without worrying about the system taking any conflicting actions.

Figure 1 outlines our system architecture. As a result of this design, each server will be able to decide on how to evict pages without any dependency on the other servers. The model for page eviction comes from an offline learning module that provides an ML model which each of the cache servers uses to determine the entry that has to be evicted. Once evicted, the system saves the (Key,Value) pair in a MongoDB instance for persistence storage.

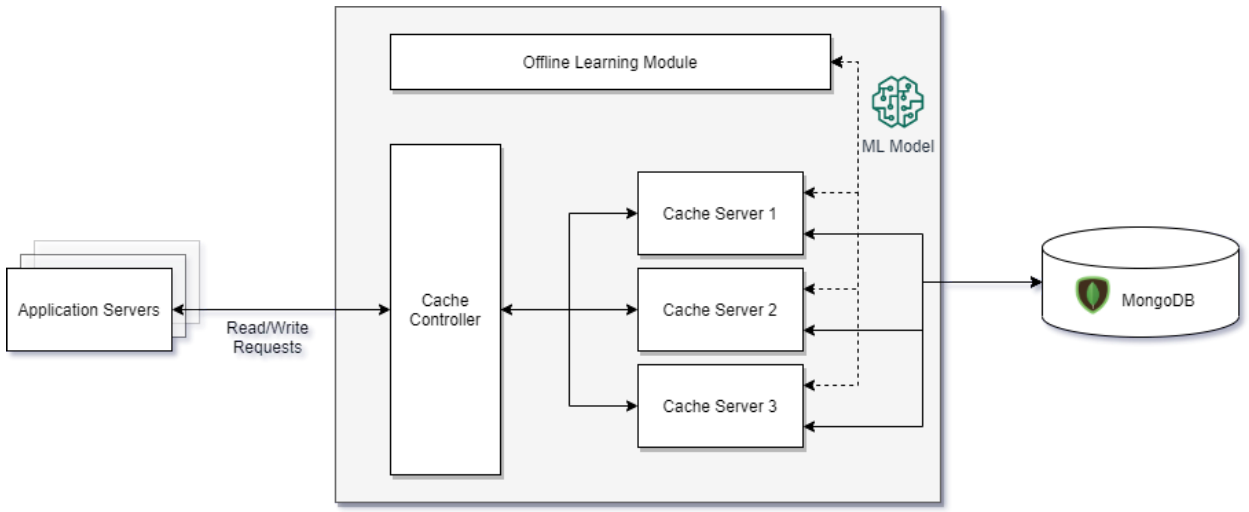

Figure 1: Cache Architecture

The system is implemented in Java using the Springboot framework. The framework enables the controller and the server to set up a RestAPI for listening to a request. The offline learning module is developed in Python and hosted using a Flask server. It exposes a Rest API which the server hits to send data for the model training. The module responds back with an index that the server uses during eviction.

4.2 Control Flow

The controller exposes an API that the application servers use to communicate with the in memory cache system. Each cache server handles a part of the key space and function independently of each other. The controller uses a hash function based on the key to determine which cache server will handle the request and forwards it to that server.

In the case of a cache hit, shown in figure 2, the cache entry is updated with the latest timestamp and its frequency is incremented. If it is a GET request, the value corresponding to the key is returned back to the user through the controller. If it is a PUT request, an acknowledgement is sent back to the user after updating the entry with the new value.

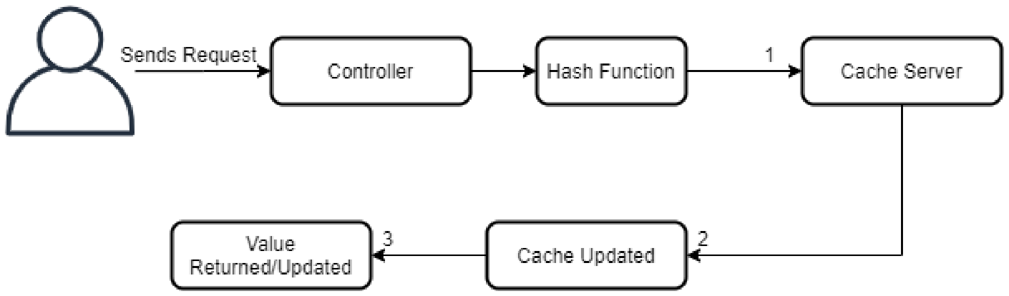

Figure 2: Control Flow During Cache Hit

On the other hand, if it's a cache miss, figure 3, the server uses a smart policy in order to evict an entry. The learning module uses a ML driven policy to determine which entry has to be evicted (6.2). This module is hosting using a flask server. The cache server sends a snapshot of the cache, containing the list of keys, its recency and frequency to the learning module, to the learning module. The module returns the index of the entry that needs to be evicted. The server then contacts the MongoDB instance to store the entry that was evicted. The response is sent back to the user similar to what is sent during a cache hit.

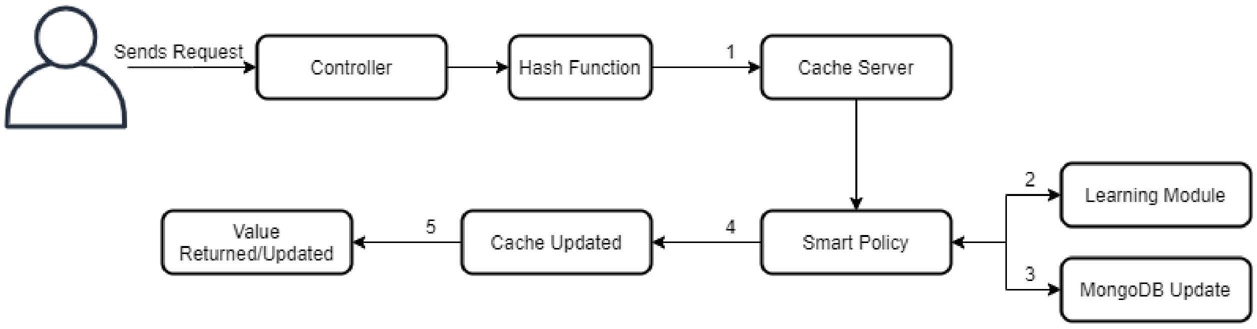

Figure 3: Control Flow During Cache Miss

5 Cache Eviction Policies

In the past, LRU and LFU cache replacement policies have proved to be more efficient than the others [7]. We propose leveraging a combination of these policies similar to Vietri et al. [7], and build an offline ML model to choose eviction policies based on the previously received page requests.

5.1 Belady's Optimal Cache

Belady cache eviction policy is the most optimal policy. In case of cache miss, the page or cache entry that is going to be requested farthest from that timestamp is evicted from the cache. The new entry gets placed in the cache. To find the evicted candidate pages, Belady needs to have access to future requests. Therefore, Belady is not useful in real-time as the future data is not present while handling the request.

5.2 LeCar

LeCar is a Reinforcement Learning (RL) based cache eviction mechanism. LeCar works with an assumption that for every workload there is a combination of fundamental LRU and LFU cache and that every workload can be handled efficiently with the combination of these fundamental cache policies. LeCar assigns and maintains a probability distribution of these two policies. In the case of a cache miss, LeCar randomly chooses between LRU and LFU as cache eviction policy and evicts the candidate. LeCar also maintains a FIFO history of cache eviction alongside the policy with which it got evicted. LeCar updates the weights if there was a cache miss and the entry was found in history. LeCar penalizes the weight of the corresponding policy to show regret of the decision. One thing to notice here, LeCar is not eviction policy rather it is an eviction mechanism that deploys on two fundamental eviction policies LRU and LFU to decide eviction candidates.

6 Smart Cache

In our work, we make use of different machine learning based methods to directly decide the eviction candidate. Smart cache is a cache eviction policy which suggest the best candidate to evict unlike LeCar which relies on the LRU and LFU.

6.1 Feature Space

To build a smart cache, we first decide the vital features for the input that can capture the cache state. For input, we use the cache snapshot which has information like, time, key, data. We maintain a separate list to store the frequency of the requests. We extract the relative time, key-value from the cache, and frequency from the frequency list to make a tuple for each cache entry. We then concatenate all the tuples together and convert this to a 1D vector as shown in Fig. 4. The model takes the 1D vector and outputs the cache id that should be evicted.

Smart cache aims to mimic the behavior of Belady cache i.e. the most optimal cache eviction policy. Thereby, we have used the Belady cache system output as ground truth to train the model.

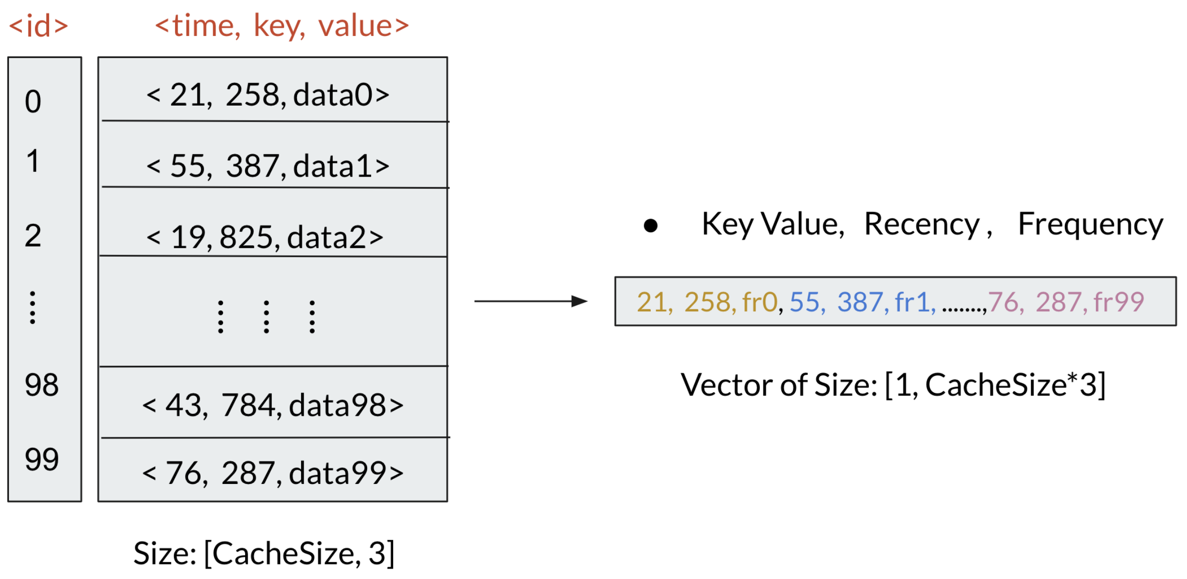

Figure 4: Feature Space

6.2 Machine Learning Models

1. **Logistic Regression**: We started with the multi-class Logistic regression. Logistic regression is a supervised classification method. Since the possible eviction index could vary from 1 to CACHE_SIZE, the output of the model should also vary in the same range. Therefore, we trained a multi-class logistic regression.

2. **Multi Layer Perceptron**: We tried with two variations of the multi-layer perceptron. One with 1 layer of 100 neurons and another with 500 neurons. We used *tanh* activation function with batch size 64.

3. **Convolutional Neural Network**: We trained a CNN model that has two convolutional layers and 2 fully connected layers. We used the ReLU activation function between the layers to introduce non-linearity. To make the model robust, we also added dropouts. To train the model, we used cross-entropy loss with Adam optimizer. The model architecture is shown in Fig. 5.

Upon new requests, the models are called if there is a cache miss and the cache is full. All the models output the cache index of the request that needs to be evicted. The evicted index will go to persistent store and will be replaced by one from the persistent store (in case of a GET), or from the user (in case of a PUT) and placed at the same index.

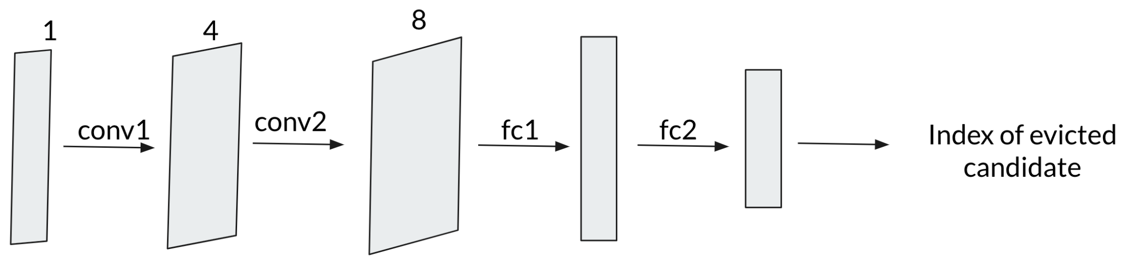

Figure 5: Convolutional Neural Network Architecture

7 Evaluation

We evaluate our implementation of the distributed cache on a synthetically generated data-set, using LRU and LFU as the baseline. Evaluations are repeated for different cache sizes and machine learning models in order to compare and contrast between them.

7.1 Test Bed

All evaluations done as part of this project were conducted on a local machine running on an Intel Core i7-9750H CPU with 12 virtual CPUs at 2.6GHz, 16GB RAM, and a nVidia GeForce RTX 2070 graphics card with 16GB totally memory. The same machine ran the Controller, Cache Servers, and the Flask server running the machine learning module.

The MongoDB persistent store was run on the cloud, with the cluster running on GCP with 3-node replication with each node have 512MB of available memory.

7.2 Data Set

A synthetically generated data-set was used for the evaluations of the baseline algorithms as well as machine learning based models. A synthetically generated was used in this project as live production traces from cache systems are hard to find as such traces invariably contain company sensitive information, or in other cases are too old [6] to use. We came across the Iotta [5] key-value traces just a few days before the end of the project and were unable to modify and complete our evaluations on this trace before the project was due. Hence, this has been left for future work.

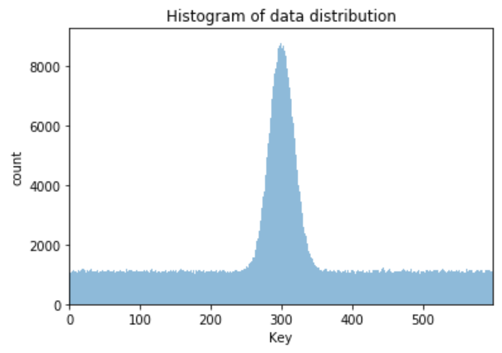

Figure 6: Key distribution for the generated training data-set

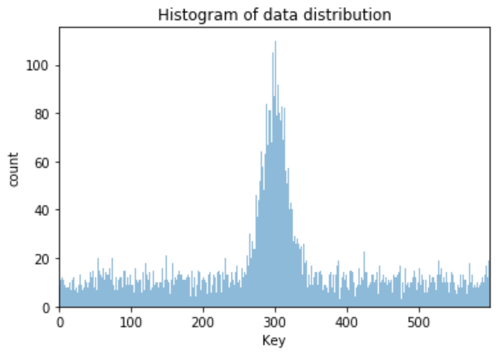

Figure 7: Key distribution for the generated testing data-set

The generated data-set that we used followed a combination of poisson and random distributions, where poisson distribution simulates an access hot-spot and the random distribution accounts for keys accessed at random throughout the life of the trace. Each key in the trace will first appear as a Write request, and the remainder of access split between Read and Write in a 75:25 ratio, simulating a 75% read oriented workload.

Two separate data-sets were generated following the described pattern, one for training and the other for testing. The training set consists of one million records having 600 unique keys. Figure 6 shows a histogram depicting the number of times each key shows up in the training data-set. The testing data-set was comparatively smaller, consists of 10,000 records with 600 unique keys, following the same distribution. Figure 7 shows a similar histogram for the testing data-set.

7.3 Results

The design of our system, and the machine learning models are such that it is data agnostic. There is also little difference between how read and write operations work in specific context of the eviction policy. Hence in our evaluations, the parameters we have worked around with are cache size, and types of machine learning model.

7.3.1 Cache Size

The results seen here are fairly straightforward and expected. As cache size increases, we find a marked improvement in the performance of models across the board. This is attributed to the larger cache size resulting in fewer evictions having to happen, boosting the hit rate.

From Figure 8, we can see that as the cache size becomes larger, the performance of all models are comparable. This validates our initial assumption that the choice of eviction policies are less important for large cache sizes, and become more important as cache sizes become smaller with respect to the number of keys being stored.

7.3.2 Choice of Model

Convolutional Neural Networks, Multi-Layer perceptrons, Logistic Regression Models were the different models that were evaluated, with LFU and LRU being the baseline. The Belady Optimal algorithms discussed earlier cannot be used in this case as it an optimal solution, not viable in actual practise. For the comparison of different models, we will focus on the results obtained with cache size 100 and 50 as the larger cache size of 200 has considerably less evictions and does not show the impact of model choice sufficiently.

Looking again at Figure 8, for cache size 100, we see that the multi-layer perceptron with 100 neurons in the hidden layer performs the best among all evaluated machine learning models, but however, while achieving better results than LRU, is still slightly less efficient that LFU. An important observation to make in this case is that simpler models like the MLP100, and Logistic Regression outperform more complicated models like CNN and MLP500 (MLP with 500 neurons in the hidden layer). This can be attributed to one of two things (or maybe both). Firstly, even with a million training data points, we might not have enough data in terms of actual evictions for the more complicated models to efficiently learn. Secondly, the patterns of the generated data might more of a simplistic nature which could be learnt better by the simpler models.

When we move over to the results of cache size 50, we continue to see the same pattern. The models we generated are able to perform comparably to LRU, but still fall short of LFU. We see a sharp fall off in the performance of the Logistic Regression model, while other models maintain the same relative level. This is probably due to the fact that the model was not able to converge with the given data.

It is clear the MLP100 model is the best performing among the lot for the given data set. It is also important to note that this might not hold true for all workloads, and a similar evaluation process to what is followed here would have to be done for any new workload considered. By looking at this data, we can come to the conclusion that there is a definite case for machine learning based eviction policies which directly predict the eviction key instead of deciding between heuristic models (as in [7]). With access to live production traces, and being able to do feature selection with respect to the workload at hand, we can definitely see the machine learning based models outperform the heuristic based policies like LRU and LFU.

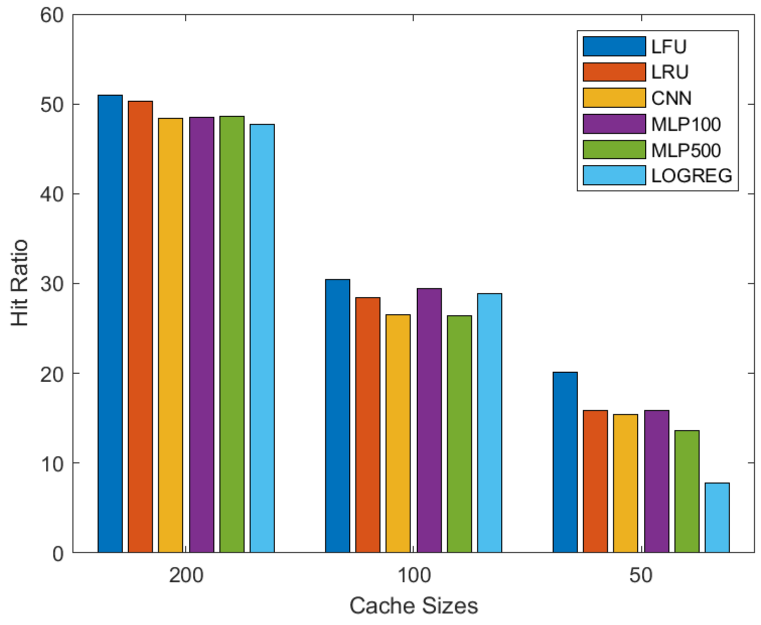

Figure 8: Comparison of models over different cache sizes

7.3.3 Normalization decisions

Another decision that evaluated was weather or not to normalize the data before training and actual use. As seen in Figure 9, we can see that the performance of the models are better across the board without normalization. Another key take away from this evaluation is that adding normalization to the process almost always adds to the overall response time of requests as an extra step needs to be performed in this case.

With this we can confidently conclude that going with out normalization would be the preferred approach for the given workload.

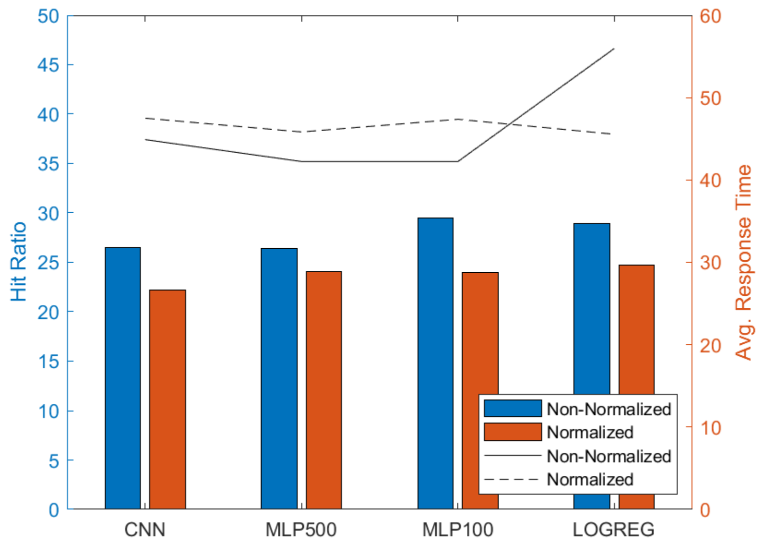

Figure 9: Comparison between normalized and non-normalized data

7.3.4 Response Times

The next statistic considered was the average response time of each request made to the system for the entire testing workload. From figure 10, we see the performance of the machine learning models are comparable to that of the baseline implementations. The slight increase for the ML models is attributed to the extra network call which has to be made to the learning module in order to make an eviction decision, as well as the slightly lower hit rate observed. We have addressed this issue with a possible solution in the Future Work section of the report.

That said, we can make the case that switching from an in-memory heuristic to a machine learning model should not see any drop in performance of the system in terms of response times of the system.

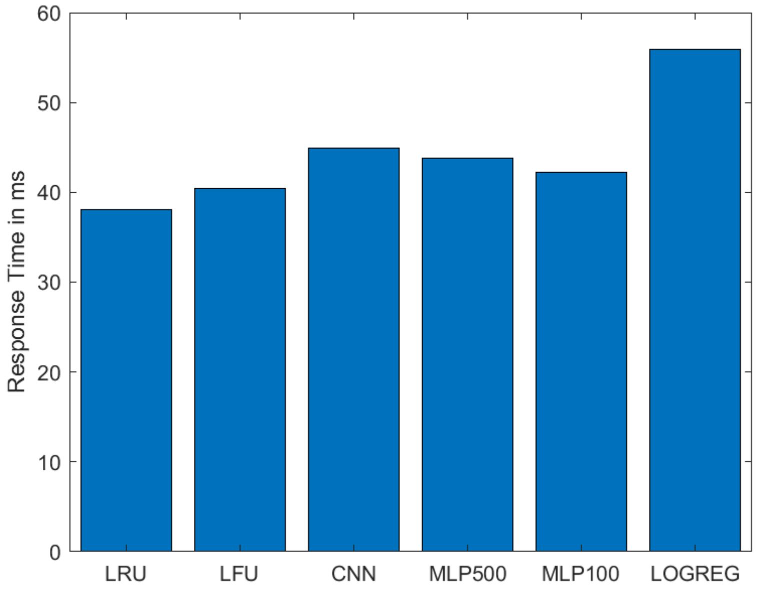

Figure 10: Comparison of average response times

8 Future Work

While our system was able to achieve comparable results to existing heuristic based policies, there are several avenues of future work that could be pursued.

Firstly, the training and evaluation of the system in this report was done on synthetically generated data. A similar evaluation can be performed with actual production traces in order to, potentially, get better performance with the help of more extensive training data, and feature modeling specific to the trace at hand. Secondly, the existing Python/Flask stack of the learning module could be converted cross-platform friendly approach like using TensorFlow and using Java Native Interface (JNI) to directly access the build models.

Another approach that can be considered is the cache being aware of "dirty" data or data that has been modified and doesn't match the state in the persistent store. With this information, the eviction policy can prioritize non-dirty data as these would not need an extra call to the persistent store. Finally, an automatic re-training process can be included into the system which can be set to trigger when the hit-rate values fall below a set threshold.

9 Division of Labour

While most tasks were a shared effort made by the team, the split responsibilities of the project are given below:

1. **Bhaargav Sriraman**: Handled the implementation of the Cache Server, the heuristic based eviction policies for evaluation, and the learned eviction policy stub.
2. **Mandakinee Singh Patel**: Handled the building and training of all machine learning modules evaluated.
3. **Suresh Siddharth**: Handled the implementation of the Cache Controller module and conducting all evaluations of the system.

10 Conclusion

As data sizes continue to shoot up, there is a need for caches to become more efficient in their eviction policies. This holds specially true in the application domain where caches are used for a multitude of purposes. This report presents a system that uses machine learning based policies to make eviction decisions. Different models with differing design decisions are evaluated and is shown to have comparable performance to existing policies. We are very confident that this approach can be taken forward to stage where it outperforms the existing models.

References

[1] Windsor W Hsu, Alan Jay Smith, and Honesty C Young. "The automatic improvement of locality in storage systems". In: *ACM Transactions on Computer Systems (TOCS)* 23.4 (2005), pp. 424–473.

[2] Samee Ullah Khan and Ishfaq Ahmad. "Comparison and analysis of ten static heuristics-based Internet data replication techniques". In: *Journal of Parallel and Distributed Computing* 68.2 (2008), pp. 113–136. ISSN: 0743-7315. DOI: `https://doi.org/10.1016/j.jpdc.2007.06.009`. URL: `http://www.sciencedirect.com/science/article/pii/S0743731507001153`.

[3] Nimrod Megiddo and Dharmendra S Modha. "ARC: A Self-Tuning, Low Overhead Replacement Cache." In: *Fast*. Vol. 3. 2003. 2003, pp. 115–130.

[4] *Memcached Architecture*. URL: `https://memcached.org/about`.

[5] *SNIA IOTTA Key-Value Traces*. URL: `http://iotta.snia.org/traces/390`.

[6] *TPC-W Homepage*. URL: `http://www.tpc.org/tpcw/`.

[7] Giuseppe Vietri et al. "Driving cache replacement with ml-based lecar". In: *10th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*. 2018.