ML-LIRS: Leveraging Machine Learning to Improve the LIRS Replacement Algorithm

Robert Fabbro*, Chen Zhong*, Song Jiang*
University of Texas at Arlington Arlington, TX
{robert.fabbro, chen.zhong, song.jiang}@mavs.uta.edu

***Abstract*—While the LIRS replacement algorithm is more capable at exploiting locality of block accesses than LRU by using its inter-reference recency (IRR) locality measure, it may still make mistakes in its decision-making thereby mis-evicting blocks from the cache. By leveraging machine learning techniques, LIRS can be improved to make more accurate decisions in situations where the IRR is untrustworthy. The use of machine learning allows for the algorithm to be more understanding of a changing access pattern to improve prediction accuracy. This leads to a lower miss ratio over the life of a workload as the algorithm will be able to pick up both longer-term and short-term patterns that a static algorithm cannot.**

***Index Terms*—Machine Learning, Cache Replacement**

I. Introduction

A replacement algorithm is a performance-critical component in a computer system. By choosing the right data for replacement, it allows more valuable data (in terms of their future access probability) in the cache for a higher hit ratio.

A. LIRS and its Weakness

LIRS is a replacement algorithm that has received much attention in the research community and industry since its inception. It has a strong base assumption that allows it to consistently attain high hit ratios in many different workloads due to its ability to effectively exploit locality while maintaining the efficiency of LRU. Specifically, LIRS uses IRR (Inter-Reference Recency) as a measurement, which is the number of distinct blocks referenced between a block's two consecutive references. It maintains a LIR stack to track recently accessed blocks similar to that of the LRU stack in the LRU algorithm. The difference from the LRU stack is that LIRS distinguishes blocks in its stack into two categories according to their IRR rankings. Blocks in the stack with low IRRs, named LIR blocks, are considered hot. Blocks of high IRRs, named HIR blocks, are cold. All hot blocks are cached. But only a fraction of cold blocks are cached. LIRS assumes that a LIR block's next IRR is also small, so that the block continues to be a LIR block. If this assumption for a block turns out not true, the block becomes a HIR block.

LIRS performs far better than LRU in the majority of cases, as the assumption usually holds. However, there are exceptions where the assumption is invalid and its locality measure, the IRR, is misleading such as in more dynamic workloads whose blocks are accessed irregularly. In such a case, LIRS may misclassify a block state (LIR or HIR) and perform poorly. While we still maintain that all blocks with low IRRs within the stack are hot by default, we incorporate a learning algorithm that incorporates an LRU window to weigh top stack hits higher. By using machine learning we can detect and correct these potential errors made by LIRS, while maintaining the core tenets of the LIRS algorithm that allows it to achieve high hit ratios. We will take the base assumption of LIRS and build upon it by adding intelligence to its decision-making process to complement its weakness.

∗ Corresponding author.

B. Why LIRS

We chose to modify LIRS as it has strong core principal on effective block retaining and replacement. Within its decision-making framework, there are some clear spots that we were able to insert a model. Its high base hit ratio was important as we wanted to make the algorithm smarter in its weak points to round out the already strong base LIRS. Not only that, but the simplicity that LIRS maintains allows for more overhead of a machine learning model than heavier and more complicated replacement policies. We will add modifications to increase intelligence, while not requiring major changes to be made to the base assumptions or algorithm.

C. Basic Idea of the New Algorithm

The algorithm at its core is LIRS. We have modified it to use machine learning decision making in replacement, while still using a LIRS stack for all recently accessed blocks and a HIR stack for a small number of HIR and in-cache blocks. The major change that we applied to the LIRS algorithm is that once the model is trained, every accessed block goes through the model to determine whether it should be moved from its current position in one of the stacks or left alone. The model does this by doing a binary classification on the block as we believe that this is largely a classification problem than a regression problem. It is much easier to decide if a block is hot or cold based on its position versus determining future access time. By simplifying the problem to being a binary classification, we make the algorithm smarter.

To enable machine learning in the design, we introduce a new sub-stack within the LIRS stack (i.e., a segment of the LIRS stack). We call this the *LRU window*. This window starts from the top of the LIRS stack and maintains a cache size amount of blocks. We partition blocks into three sections according to their recency measured by the LIRS stack. The

top section is from the stack top to the LRU boundary, which is the LRU window. The second section ends at the bottom of the LIRS stack, named *LIRS window*. This bottom is marked by the LIR block of the largest recency. The last section includes all remaining blocks that are out of the LIRS stack. In addition, there are two other features used to effect weighting in this position feature. One is the aged count of accesses a block has. The other is whether an accessed block was designated HIR previously. These two features help give the model a better glimpse into what blocks are more important than others. By using these features, our algorithm can consider IRR whenever it is applicable in a workload maintaining LIRS strengths. In the event where the IRR becomes unstable or unusable, the new algorithm can detect that and choose LRU where it is stronger. This ultimately creates an algorithm that is as accurate as LIRS in the majority of cases, while being more accurate in cases exhibiting its weakness.

We name the LIRS algorithm optimized by machine learning ML-LIRS. Next, we will explain the general workings of the algorithm and assumptions. We will go further into depth on how the LRU stack works, the model selected, training methodology, and finally the decision-making placement.

II. The ML-LIRS Algorithm Design

There are multiple components that allow for ML-LIRS to effectively function, including feature gathering, training, model, and decision-making process. In the following subsections we discuss each component of the algorithm.

A. The LRU Window

As we have indicated, the LRU window is a sub-stack that is cache-size long at the top of the LIRS stack. Its purpose is to mimic what a standard LRU cache would do temporally. We have the bottom block of this window be a boundary. The boundary initialization occurs when the LIRS stack becomes full of resident blocks. Once it is full, the bottom-most block is set as the LRU boundary. When a block is added at the top this boundary moves up to maintain a cache size amount of blocks. This window remains at the top, thereby satisfying our purpose of opportunistically giving importance to accesses at the top of the stack versus the bottom.

The reason for using a cache sized sub-stack over any other size is because one of the faults we have determined of LIRS is that it always assumes that IRR can consistently make accurate predictions. However, there are workloads where IRR can fail to be a predictor, and therefore accesses at the top of the stack with a lower recency need to be treated as more important than LIR ones at the bottom. Essentially, there are times where taking a temporal approach is better. What our model does is determine which position is hotter during the accesses by using the other two features given. We attempt to gradient the measure of hotness of any individual block via learning methods to choose whether LIRS is preferable or LRU.

B. The Features

To allow the model to make good decisions we feed it three features to exploit both longer-term and short-term locality.

The first feature is the position of where the block was accessed. The positions that the block can be in are in the LRU window and in the LIRS stack, not in the LRU window but in the LIRS stack, or outside the LIRS stack. Each position is weighted the same. When a block is accessed, its position feature is marked as one of the three possible positions accordingly. This position feature allows the model to weigh different sections of the LIRS stack differently. If there are more accesses at the top of the stack, the model will favor that position. This lets the top of the stack become more important than accesses in the bottom. If this isn't true for a particular trace, the model will adapt. In contrast, LIRS does not differentiate accesses occurring in different positions of the LIRS stack.

The second feature is a weighted access count. LRU considers one past access to decide if a block is hot. And LIRS considers two past accesses. Both of them do not take longer-term history accesses into account, which may be necessary for access patterns with constant short-term variations. To this end, ML-LIRS tracks access count for each block in the LIRS stack. To keep frequently accessed blocks from being not responsive to recent access events, it incrementally phases out impact of history accesses on the count value. Specifically, ML-LIRS maintains a global clock, which ticks after every period of N accesses. A block also has its timestamp, which is updated to the current global clock time upon an access. When a block is accessed, its weighted access count is updated as $count \times \alpha^{(clock-timestamp)} + 1$, where $count$ and $timestamp$ are the count value and timestamp, respectively, immediately before the access, $clock$ is the current clock time, and α is the aging rate (e.g., 90%), a percentage indicating how fast history accesses should be phased out with each clock tick.

The third feature is the block's last HIR designation status. We still work under the assumption that if a block has a HIR status then it is cold and if it is LIR, hot. If the block had the HIR status when it was last accessed, then that means it was considered to be cold in the past. By letting the model know the block's past designation, it can make a better decision based on its new access count and current position.

C. The Model

We have tested multiple different machine learning models, including Naive Bayes Binomial [3], Naive Bayes Multinomial, Perceptron [7], and Logistic Regression [5]. Among the four, the Naive Bayes Multinomial model gave the best performance. The perceptron model had too much variance and inaccuracy over the lifetime of the workload due to low training data size and fast-changing access patterns. Logistic Regression proved to also provide poor performance. The Naive Bayes Binomial model was the second model that we selected and ran many live trace tests on, giving high LIRS performance on most workloads, but wasn't optimal due to our discrete data. This is the reason why we selected the Naive Bayes Multinomial model as our features are discrete, which the Multinomial model excels with [5].

The chosen model modifies the LIRS decision-making pro-

cess through the use of the stated features. Taking the inputted features into account, the Multinomial model outputs probabilities that are cutoff to either being hot at "1" or cold at "0". We use this probabilistic decision process based on previous training to attain higher accuracy in block classification. Using a probabilistic model is optimal as cache decision-making is probabilistic at its core as many variables can change over the course of a workload or an actual live service.

D. Labeling and Training

Labeling of data happens when a block is accessed and has at least one previous access. When the block satisfies the criteria, we check if the block is in the LIRS stack. If it is, we label the data as hot, or "1" as a label. If it is outside of the stack it is labeled as cold, or a "-1" label. The labeling and gathering of data happen at any access in a workload.

Training happens at a pre-determined size. The optimal size found from experimentation, was at 1000 reference blocks in the training data set. This is the size that allows for the algorithm to almost always follow LIRS when it works best and catch most irregular IRR moments. When the training amount is hit, we train the model using the partialfit function that is included with the Scitkit Learn's library [5]. The partialfit function is used as it allows the model to continue iterating over itself with new data rather than overwriting itself, thereby allowing it to learn the general access pattern. One issue that should be noted is that if the workload suddenly has a change in the access pattern, the model will suffer until it can fit a few times and learn the new access pattern.

After the training is complete, we can start using the model. This is a parameter that can be changed if needed, however, we found that setting it to a start of 3000 blocks of training to be optimal. Once it begins it flips from LIRS block classification decision to the model's.

Algorithm 1: Decision Making and Placement

```
Upon access of Block B
Fetch the feature of B including access position, B's HIR status and
  normalized acces.
Use the predictor to predict the B's HIR/LIR status
if B is a HIR block currently then
    if B predicted as LIR then
        Promote it a LIR block;
        Demote the LIR block currently at the bottom of the LIRS stack to
          the Q stack;
        Keep removing any block at the bottom of the LIRS stack until a
          LIR block is at the bottom.
    else
        Add B to Q stack
    end
else
    if B predicted as LIR and LIR size > HIR size then
        Add B to Q stack
        if B is at the bottom of LIRS stack then
            Promote it a LIR block;
            Keep removing any block at the bottom of the LIRS stack
              until a LIR block is at the bottom.
        else
            Promote it a LIR block;
        end
end
Insert B into the LIRS stack at its top;
```

E. Decision Making and Block Placement

The decision-making section of the algorithm is simple. As one can see from the pseudo code 1, on each block access, the model runs a prediction using the features the block has. If the block is currently designated HIR, and if the prediction is that it is hot, we switch it from HIR status to LIR status, and prune the stack like LIRS normally does. If it is predicted as cold, nothing is done. If the accessed block is already in the LIRS stack and the model predicts it to be cold, we move it from the LIRS stack to the HIR stack. This is the only deviation from the LIRS way of moving blocks from LIR to HIR. In LIRS, the only block in the LIRS stack that can change designation from LIR to HIR, is the bottom-most block with the highest IRR [2]. Our modification is that any block within the LIRS stack can change designation if the model decides it should, circumventing the requirement that IRR is the only means of deciding if a block is hot or cold. Finally, if the accessed block is in the LIRS stack and is predicted to still be hot, it remains as is.

III. Evaluation

In this section we will evaluate the performance of ML-LIRS in relation to LIRS and LRU, and demonstrate strengths that machine learning provides, and analyze its limitations.

A. Experiment

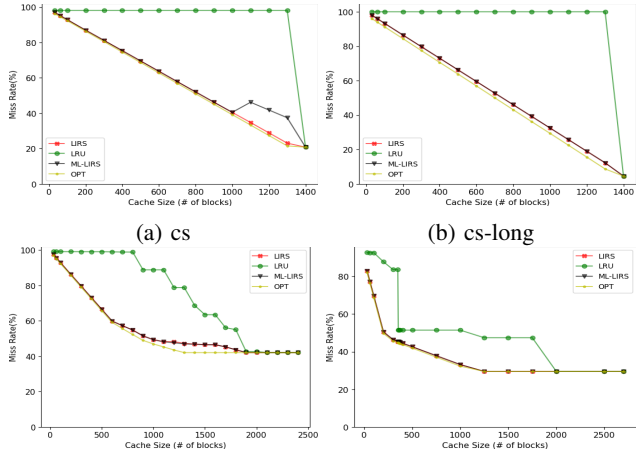
In the experiments we adopted the same traces used in the original LIRS paper: *2-pools*, *cpp*, *cs*, *glimpse*, *postgres*, *sprite*, *multi1*, *multi2*, and *multi3* [2]. Along with this, we added in a few additional traces that have untrustworthy IRRs in order to get a better understanding of ML-LIRS's behaviors compared to the base LIRS algorithm. The new traces are as follows:

1) **cs-long** is a longer version of cs. It is a C source program examination tool trace with 300,800 references.
2) **zigzag** is a trace with unstable IRRs. References are in a zig-zag pattern going back and forth (152,001 references).
3) **zigzag-cs-hybrid** is a combined trace of *zigzag* and the cs trace. The length of this trace is 306,078 references.
4) **zigzag-2-pools** is a combined trace of *zigzag* and 2-pools with 152,001 references.
5) **zigzag-sprite** is a combined trace of *zigzag* and sprite with 185,997 references.
6) **zigzag-multi1** is a combined trace of *zigzag* and multi1 with 67,859 references.
7) **zigzag-multi2** is a combined trace of *zigzag* and multi2 with 78,312 references.

B. Performance in Looping Workloads

Fig. 1 shows miss ratio curves of *cs*, *cs-long*, *gli*, and *ps* with LIRS, ML-LIRS, LRU, and OPT. As can be seen, LIRS performs incredibly well in all of these traces closely following OPT, with very little deviation except for *cs*.

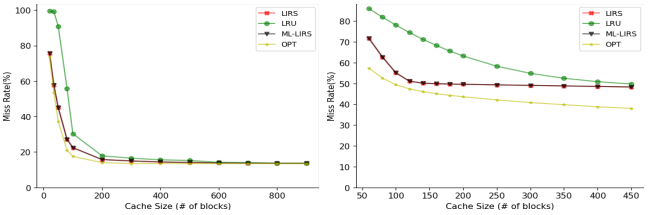
At the three higher cache sizes, there is an anomaly that happens where the LRU window is just large enough where enough hits happen in the window that confuses the model. This makes the model weigh the LRU window higher where it



(a) cs (b) cs-long

(c) gli (d) ps

Fig. 1: Miss ratio graphs for *cs*, *cs-long*, *gli*, and *ps*



(a) cpp (b) 2-pools

Fig. 2: Miss ratio graphs for *cpp* and *2-pools*

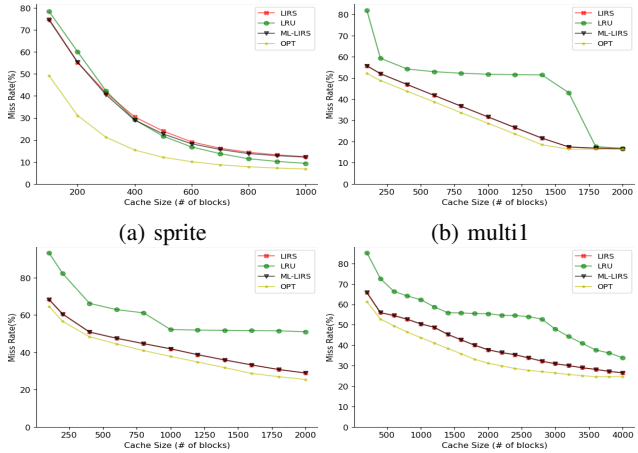
disconnects from LIRS at certain points in the trace thinking LRU is more accurate. Though despite this, it does correct itself as it stays closer to LIRS than LRU. A solution would be to increase training intervals to catch these types of errors at a performance cost.

C. Performance in Probabilistic Workloads

Fig. 2 shows the miss ratio curves of *cpp* and *2-pools*. *Cpp* is far more probabilistic than *2-pools*. We see that ML-LIRS has no issues with probabilistic workloads and can get close to OPT due to LIRS's strong base assumptions. A strength of ML-LIRS is that in workloads where LIRS is generally better, it will side with it. The only times of confusion will be where many hits appear that favors the opposite replacement policy and it changes causing a worse short term hit ratio.

D. Performance in Temporally Clustered Workloads

In Fig. 3a we have the *sprite* trace whose accesses are clustered on a limited number of blocks and exhibit strong locality. This trace works very well for LRU as many of the reference blocks are accessed within the LRU window and rarely outside. Our algorithm is able to follow LIRS and even improve on LIRS at the middle cache sizes as it follows LRU for a bit. However, at the higher cache size our algorithm ends up matching LIRS. This is due to a few reasons. LRU benefits because the time between accesses fits within the cache at the larger sizes, so its temporal approach works. The LRU sub-stack attempts to mimic this, but it holds non-resident blocks as well as resident, meaning that the LRU sub-stack is more



(a) sprite (b) multi1

(c) multi2 (d) multi3

Fig. 3: Miss ratio graphs for *sprite*, *multi1*, *multi2*, and *multi3*

often less than what a true LRU stack would be. Due to this, there are hits that happen in the upper and bottom LIRS stack causing confusion on which is more important.

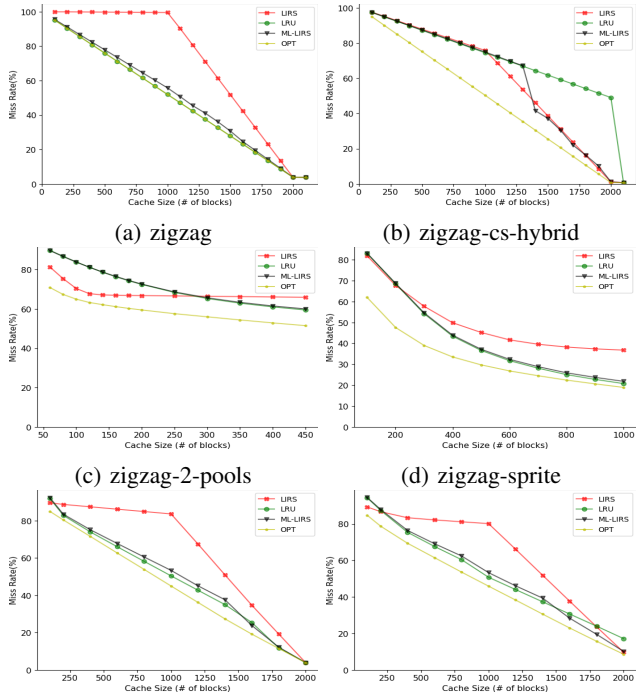
This is a problem that can occur if there are moments where even the position of access isn't useful. The model will learn which is better, even if slight, and go with that given the other two features. This sometimes benefits the algorithm, but sometimes it can choose the sub-optimal replacement policy.

E. Performance in Mixed Workloads

Fig. 3 shows the miss ratios for the three workload traces, *multi1*, *multi2*, and *multi3*. The mixed workloads exemplifies LIRS strong points, which ML-LIRS makes little improvements on. ML-LIRS performs equal to LIRS and is able to avoid selecting LRU as a better algorithm. While there are accesses within the LRU window, our model is able to understand that those are unimportant due to how few the event occurs and weighs LIRS strongly where it does not deviate at all. ML-LIRS is intelligent enough to not deviate due to a small number of LRU accesses, making it fairly robust to small changes.

F. Performance in Untrustworthy IRR Workloads

In Fig. 4 we show multiple different experiments that have an IRR that keeps changing and is not an unreliable predictor of future accesses. The experiments all have a section where the IRR is unreliable. Except for *zigzag*, all these traces are combinations of the traces in the prior sections to exemplify how a section in any trace where the IRR is unreliable can impact LIRS and ML-LIRS. In *zigzag* we see ML-LIRS outperforms LIRS significantly as LIRS stays stagnant throughout majority of the trace, while ML-LIRS improves with each cache size increase. A majority of IRRs produced in *zigzag* are an unreliable one, which explains LIRS's weak performance. In *zigzag-cs-hybrid* we see LIRS performing better than ML-LIRS as it can detect the locality change from the zigzag section and start improving the miss ratio when it gets to the *cs* section. ML-LIRS takes longer to adjust.



(a) zigzag (b) zigzag-cs-hybrid

(c) zigzag-2-pools (d) zigzag-sprite

(e) zigzag-multi1 (f) zigzag-multi2

Fig. 4: Miss ratio graphs for *zigzag*, *zigzag-cs-hybrid*, *zigzag-2-pools*, *zigzag-sprite*, *zigzag-multi1*, and *zigzag-multi2*

Zigzag-2-pools has LIRS performing better overall, but is improved upon at higher cache sizes. In this experiment, it seems that ML-LIRS was not able to take advantage of locality as enough hits in the LRU cache leaves it unable to change its weights in any meaningful way. *Zigzag-sprite* has ML-LIRS showing an improvement early on. In an unreliable IRR setting, it generally follows LRU, and in this case, it follows LRU down to almost being OPT.

In the two zigzag-multi experiments, we see an improvement over LIRS. In *zigzag-multi1* and *zigzag-multi2* we see ML-LIRS following LRU closely and improving while LIRS is mostly stagnant for some time before beginning improvement about halfway into the curve. The zigzag-multi experiments show the potential performance improvements ML-LIRS can provide for a real-world workload. If a trace has an unreliable IRR and multiple different patterns within it after or before, the algorithm can adapt and improve while LIRS struggles.

IV. Related Work

Cache replacement is one of the most fundamental problem in a caching system. Learning-based improvements on traditional replacement algorithms have been proposed recently.

Some learning based solutions [1], [11] learn from past caching behavior to predict future pattern in accesses of a CPU cache. Such a policy formulates cache replacement as a binary classification problem, where the goal is to predict whether incoming cachelines are cache-friendly or cache-averse. For example, SHiP [11] tries to observe the evictions from the cache to study the behavior of access pattern. Instead of learning the behavior of history accesses, Hawkeye [1] learns from the optimal replacement algorithm.

Machine-learning technique has been employed in the design of replacement algorithms for I/O accesses, such as Glider [8], LeCaR [10], DeepCache [4], LRB [9], and CACHEUS [6]. ML-LIRS builds on LIRS. It reconstructs the LIRS and uses the ML predictor to predict which block has higher possibility to be a hot blockt. It uses an LRU Window to try to avoid the situation that the IRR can fail to be a good predictor. With the combination of LRU window, ML-LIRS model takes the block access position into consideration, so that it is able to balance the impact of IRR and recency.

V. Conclusion

In this paper we propose ML-LIRS, a block replacement algorithm that integrates the strengths of the LIRS algorithm and LRU via machine learning. It performs as well as LIRS in traces where there is little room for improvement while performing better where LRU provides better insight. We show the weaknesses that LIRS has in workloads with unreliable IRR locality predictor where ML-LIRS aids in. Leveraging learning algorithms can allow for replacement policies to mold themselves to current workloads rather than always be rigid and set in a certain rule set, creating flexibility in decision-making. This is an area that can be further explored and possibly lead to many real-world applications.

References

[1] Akanksha Jain and Calvin Lin. Back to the future: Leveraging belady's algorithm for improved cache replacement. In *ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016.

[2] Song Jiang and Xiaodong Zhang. Lirs: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. *SIGMETRICS Perform. Eval. Rev.*, 30(1):31–42, June 2002.

[3] Andrew McCallum and Kamal Nigam. A comparison of event models for naive bayes text classification, 1998.

[4] Arvind Narayanan, Saurabh Verma, Eman Ramadan, Pariya Babaie, and Zhi-Li Zhang. Deepcache: A deep learning based framework for content caching. In *The 2018 Workshop on Network Meets AI & ML*, 2018.

[5] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[6] Liana V. Rodriguez, Farzana Yusuf, Steven Lyons, Eysler Paz, Raju Rangaswami, Jason Liu, Ming Zhao, and Giri Narasimhan. Learning cache replacement with CACHEUS. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, 2021.

[7] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 1958.

[8] Zhan Shi, Xiangru Huang, Akanksha Jain, and Calvin Lin. Applying deep learning to the cache replacement problem. In *The 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019.

[9] Zhenyu Song, Daniel S. Berger, Kai Li, and Wyatt Lloyd. Learning relaxed belady for content distribution network caching. In *17th USENIX Symposium on Networked Systems Design and Implementation*, 2020.

[10] 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.

[11] Carole-Jean Wu, Aamer Jaleel, Will Hasenplaugh, Margaret Martonosi, Simon C. Steely, and Joel Emer. Ship: Signature-based hit predictor for high performance caching. In *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2011.