Driving Cache Replacement with ML-based LeCaR

Giuseppe Vietri[†], Liana V. Rodriguez[†], Wendy A. Martinez[†], Steven Lyons[†],
Jason Liu[†], Raju Rangaswami[†], Ming Zhao[‡], Giri Narasimhan[†]
[†] *Florida International University* [‡]*Arizona State University*

Abstract

Can *machine learning* (ML) be used to improve on existing cache replacement strategies? We propose a general framework called **LeCaR** that uses the ML technique of *regret minimization* to answer the question in the affirmative. We show that the **LeCaR** framework outperforms ARC using only two fundamental eviction policies, LRU and LFU, by more than 18*x* when the cache size is small relative to the size of the working set.

1 The Case for ML in Cache Management

Can *machine learning* (ML) be used to learn from the best cache replacement policies? This question was answered in the affirmative as far back as 2002 by the ACME system by Ari et al. [2], which assumes that there exist a pool of multiple "experts" (i.e., strategies). Each expert had an associated weight and the expert with the highest weight made the recommendation for an item to be evicted on a miss. The weight of an expert at any given time was a function of its recent performance. ACME improved on the performance of LRU and LFU. Since 2002, both cache replacement and machine learning have seen major advances (e.g., adaptive replacement [3, 24]; regret minimization and online learning [10,11,27]), suggesting a revisit of this question.

Can ML improve on existing cache replacement strategies? In this paper we answer this question in the affirmative. This is non-trivial given the diversity of data sets for which ARC has been shown to perform at or close to the best among its competitors. We identify one area where ARC may not have been adequately put through its paces, i.e., when cache is much smaller than the working set. Although ARC may outperform other algorithms even for small cache sizes, when a "stable" working set does not fit in cache, it suffers a loss in performance (referred to as the proverbial "ARChilles' Heel" (sic) [29]), leaving room for further improvements.

Can we design "scalable" ML solutions for cache replacement? Scalable solutions can maintain optimal performance regardless of workloads. It has been shown that working sets are telescoping in nature with larger working sets fully containing one or more smaller working sets, motivating the utility of caches at various granularities [20, 21]. Although a one-time correct sizing of cache memory is not a trivial problem, it is possible to provision sufficient cache memory relative to the entire working set in order to maintain acceptable performance. However, this may not be achievable because (a) caches are expensive; (b) small caches are par for the course (e.g., in mobile and IoT devices); (c) workloads are highly consolidated and dynamically change over time, and it may be unaffordable to provision for peak workloads; (d) it may be impossible to anticipate all the possible workloads the system would have to face in the near future; and (e) reprovisioning hardware caches is either infeasible or not straightforward. Thus, we need caching algorithms that scale well when workloads get larger relative to cache sizes.

Figure 3 shows that our ML-based **LeCaR** (**Le**arning **C**ache **R**eplacement) is competitive with ARC for relatively large cache sizes, but is markedly superior to it when cache sizes become smaller. Given the "online" nature of the cache replacement problem, we use techniques from the subarea of *online learning* with regret minimization [5, 10, 11, 23, 27, 31]. **LeCaR** outperforms ARC by as much as 18*x* in hit rate across the 8 production storage I/O workloads from FIU [26] when caches are set to 0.1% of workload size and up to 30.5% when caches are at 1% of workload size. For larger caches **LeCaR** performs competitively relative to ARC.

2 Motivation

"Single gear" strategies are limited. The best-known strategies for cache replacement are LRU and CLOCK [9], both of which tend to retain pages with high recency, and LFU, which retains pages based on how frequently they have been referenced. These *static* strategies cannot adapt to changes in workloads and fail to have good all-round performance, especially when recent pages are not frequently accessed or when pages are accessed a number of times and then lapse into long periods of infrequent access. In practice, LRU can evict frequently accessed pages, while LFU can evict fairly recent pages while "hoarding" entries that were frequently accessed in the distant past.

"Adapting" to the road. A common theme in most improved algorithms is that they *combine* actions based on *recency* and *frequency*. A spectrum of *adaptive* algorithms combining the strengths of LRU, LFU, and/or CLOCK have been proposed. These include LRFU [22], DUELINGCLOCK [15], LRU-*K* [25], LIRS [17], CLOCK-PRO [16], 2Q [19], and more. The best among these were the two adaptive algorithms called *Adaptive Replacement Cache* (ARC) [24] and CLOCK *with Adaptive Replacement* (CAR) [3]. The core idea behind ARC and CAR was that they separated the recent pages accessed only once from the frequent pages into two partitions of the cache, and used clues from a limited set of history pages to decide the relative sizes of the partitions. The two partitions were managed as FIFO queues and eviction decisions were made using a complex set of deterministic conditions. An alternative view is that ARC and CAR reduced eviction to a choice between LRU and a version of LFU that does not differentiate between entries that are accessed more than twice.

Driving down Machine Learning avenue. Theoretical research is limited in its ability to distinguish between LRU and ARC (and CAR as well) [8]. Researchers continue to actively pursue ways to improve the algorithms. A natural question is: can *Machine Learning* and related predictive techniques help to improve cache replacement algorithms. Predicting branchings [18] and time before re-referencing [13, 14] have been tried; other soft computing techniques such as *Neural Networks* (NN), *genetic algorithms* (GA), *Classification and Regression Trees* (CART), *Multivariate Adaptive Regression Splines* (MARS), *Random Forest* (RF) and *TreeNet* (TN) have also been used for this problem [1, 7, 28, 30] with limited success. Adaptive caching with ML was attempted [2,12] and they are known to outperform only the "static" strategies, LRU and LFU. Also, they are expensive to implement and were not pursued. The main goal here was to overcome the cost of implementation while outperforming more advanced algorithms developed since that work.

Turning into "online learning" lane. An appropriate model of machine learning for cache replacement is that of *Reinforcement online learning* (RL), which require that decisions be made *online* as requests arrive and that cumulative *regret* (or *reward*) be optimized. Reinforcement online learning constantly acquires new knowledge and adapts to changes in input characteristics [11,23,31]. Cache replacement has been modeled as a *Multi-Armed Bandit* (MAB) problem [6, 10] for highly specialized caches, but has not been adequately evaluated in standard settings [4]. **LeCaR** explores online learning with variants of MAB.

This brings us to the proposed ML-based algorithm called **LeCaR**, which uses reinforcement online learning with regret minimization. It is important to note that the regret minimization approach allows room for theoretical guarantees of performance to go with its performance in practice. However, the theoretical framework is different from a regular online learning method because the feedback about the quality of the decision made at any given time is delayed and not instantaneous.

The high octane fuel. While the online learning provides the scaffold, the secret of success of the **LeCaR** framework lies in "combining" two fundamental, yet orthogonal, policies – *recency* and *frequency*. Even though LFU is much reviled and regularly puts up poor performances in its pure form, its contribution (weight) as managed by **LeCaR** is often very high, suggesting that LFU is a blunt but powerful tool that needs a regular infusion of cleanup best provided by the orthogonal recency policy. **LeCaR** allows items with high frequency to be efficiently evicted based on their recency. Surprisingly, using LFU with decay instead of pure LFU lowers the efficacy of **LeCaR**. Furthermore, replacing LRU with ARC also lowers the efficacy of **LeCaR**. One possible explanation is that both ARC and LFU with decay tamper with pure LFU, destroying its orthogonality from native LRU.

Our work also suggests that for every workload there is some combination (i.e., a probability distribution) of LRU and LFU that can handle it as well as any other deterministic cache replacement scheme. ARC may be combining recency and frequency in artful ways, but does not have access to the full probability distribution that our proposed **LeCaR** framework has.

3 Peeking under the hood of LeCaR

Unlike ACME, we do not assume that our system has access to a set of best strategies (i.e., experts). Note that simulating a collection of experts can be an expensive proposition. Instead, **LeCaR** assumes that at every instant, the workload is best handled by a judicious "mix" (i.e., a probability distribution) of only **two fundamental policies**: *recency*-based and *frequency*-based evictions. Thus the goal of **LeCaR** is to "learn" the "optimal" probability distribution for every state of the system. Unlike ACME, **LeCaR** maintains a probability distribution of two policies instead of a probability distribution of a panel of expert strategies. Surprisingly, this *minimalist* approach to learning achieves outstanding results.

Another distinguishing feature of our system is that the weight associated with the two policies is not a function of their current hit rate, but of the current associated *regret*. Thus, we model cache replacement as an *online learning* problem involving *regret minimization* [31]. To handle any cache miss, one of the two policies is chosen at random (probabilities derived from their associated cumulative regret values due to the misses they "caused").

In order to manage regret, the cache manages a FIFO

Algorithm 1: LeCaR(LRU,LFU)

```
Input: requested page $q$
if $q$ is in $C$ then
    C.UPDATEDATASTRUCTURE($q$)
else
    if $q$ is in $H_{\mathbf{LRU}}$ then
        $H_{\mathbf{LRU}}$.DELETE($q$)
    else if $q$ is in $H_{\mathbf{LFU}}$ then
        $H_{\mathbf{LFU}}$.DELETE($q$)
    UPDATEWEIGHT($q, \lambda, d$)
    if (Cache $C$ is full) then
        action = (LRU, LFU) w/ prob $(w_{\mathbf{LRU}}, w_{\mathbf{LFU}})$
        if (action == LRU) then
            if $H_{\mathbf{LRU}}$ is full then
                $H_{\mathbf{LRU}}$.DELETE(LRU($H_{\mathbf{LRU}}$))
            $H_{\mathbf{LRU}}$.ADD(LRU($C$))
            $C$.DELETE(LRU($C$))
        else
            if $H_{\mathbf{LFU}}$ is full then
                $H_{\mathbf{LFU}}$.DELETE(LRU($H_{\mathbf{LFU}}$))
            $H_{\mathbf{LFU}}$.ADD(LFU($C$))
            $C$.DELETE(LFU($C$))
    $C$.ADD($q$)
```

history of metadata on the most recent evictions from the cache. When an entry is evicted from the cache it is moved to history. The number of entries in the history (as with ARC) is equal to the number of entries in the cache. Each history entry is labeled by the policy that evicted it from the cache. A decision is considered "poor" if a request causes a miss and if the requested page is found in history. The intent is that a miss found in history could have been rectified by a more judicious eviction, and hence the regret. Regret is graded and is larger if it entered the history more recently. When poor decisions are caused by a specific policy, that policy is penalized by increasing the "regret" associated with it. (See Algorithm 1.)

Algorithm 2: UPDATEWEIGHT(q, λ, d)

```
Input: page $q$, learning rate $\lambda$, discount rate $d$
$t :=$ time spent by page $q$ in History
$r := d^t$
if $q$ is in $H_{\mathbf{LRU}}$ then
    $w_{\mathbf{LFU}} := w_{\mathbf{LFU}} * e^{\lambda * r}$ // increase $w_{\mathbf{LFU}}$
else if $q$ is in $H_{\mathbf{LFU}}$ then
    $w_{\mathbf{LRU}} := w_{\mathbf{LRU}} * e^{\lambda * r}$ // increase $w_{\mathbf{LRU}}$
$w_{\mathbf{LRU}} := w_{\mathbf{LRU}} / (w_{\mathbf{LRU}} + w_{\mathbf{LFU}})$ // normalize
$w_{\mathbf{LFU}} := 1 - w_{\mathbf{LRU}}$
```

Algorithm 2 calculates the weights. The weights start off being equal, although they may be initialized using some *a priori* information about the algorithms or the request sequence. When a "regrettable" miss is attributable to LRU (resp. LFU), because it is found in history $H_{\mathbf{LRU}}$ (resp. $H_{\mathbf{LFU}}$), the weight of the "other" policy, i.e., LFU (resp. LRU) is updated as recommended by regret minimization [5, 10, 27, 31]. Note that λ is the learning rate (initially 0.45), d is the discount rate (initially $0.005^{1/N}$, where N is the cache size), and reward, $r = d^t$. Performing a sensitivity analysis proves that the algorithm is robust to the learning rate and discount rate parameter and we and chose ones that worked reasonably well across the eight workloads we experimented with. Finally, the algorithm returns the weight of LRU and LFU, which are used to choose one of the two policies to apply probabilistically for the next miss.

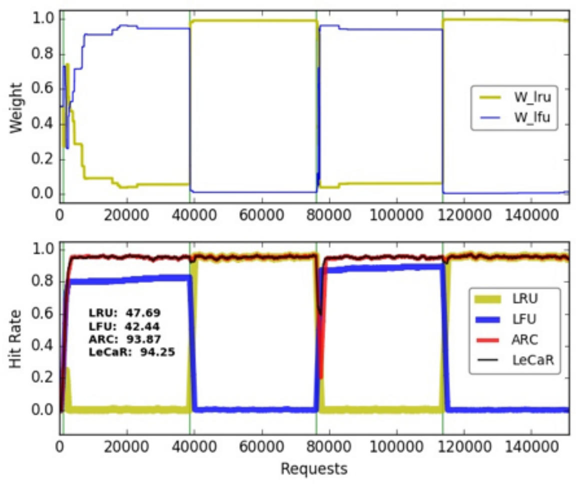

Figure 1: Switching between favorable sequences

4 Evaluation

Here we present a series of experiments designed to argue the viability of the ML framework presented above.

Does LeCaR learn? How quickly? We start with some simple experiments using synthetic data to establish **LeCaR**'s ability to learn. We synthetically generated sequences that periodically switch back and forth from being favorable to LRU to being favorable to LFU. During the phase when it is favorable to LRU, it generates requests that deliberately cause a hit for LRU and a miss for LFU, and vice versa. The generator also includes a small amount (0.5%) of noise, i.e., requesting pages from the *universe* that are neither in LRU nor in LFU's cache. For these data sets, size of the cache was set at 500 entries, with a universe of pages of size 15 times the size of the cache.

Figure 1 shows a plot that is broken into four equal sections, each partitioned by blue vertical lines. Each

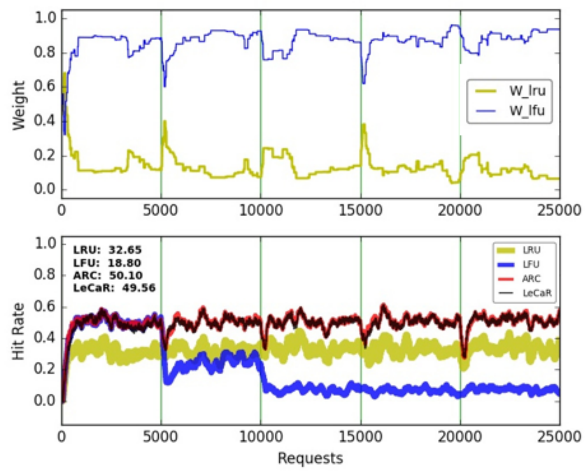

Figure 2: Impact of phase change

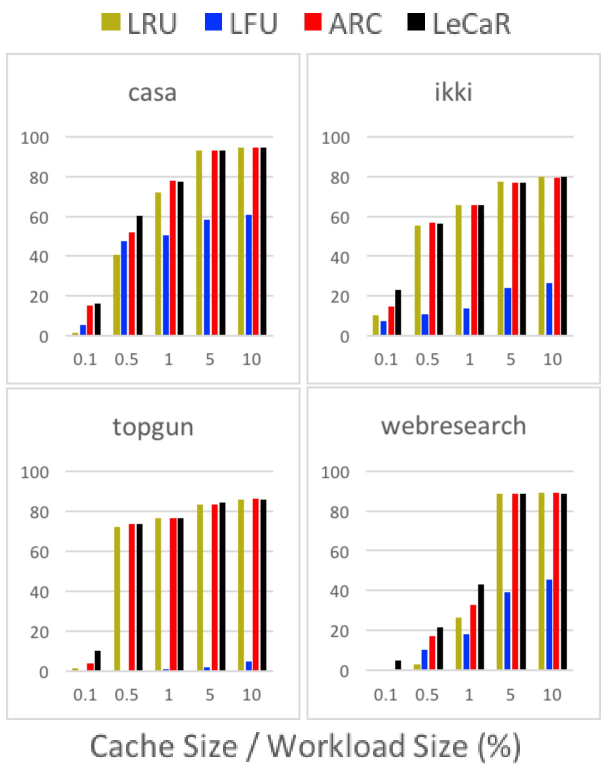

Figure 3: **LeCaR** outperforms ARC and LRU at lower cache sizes and is competitive at high cache sizes for workloads from FIU Traces. X-axis: Cache size as a fraction of the workload's size. Y-axis: Cache hit rate. Details in Sec. 4.

partition corresponds to a sequence that is favorable to one of LRU or LFU, as described above. The lower part of the figure shows the hit rates with each of the four algorithms under study. Hit rates for the four algorithms are represented by curves in four different colors as indicated. (Note that hit rates are computed over a sliding window of size 500.) The upper part of the plot shows the weights of LRU and LFU as **LeCaR** progresses through the request sequence. Note that the hit rates of ARC and **LeCaR** are very close to each other. The lower plot shows how quickly ARC and **LeCaR** adapt to the change. The learning for **LeCaR** is also reflected in fluctuations in the weights w_{LFU} and w_{LRU}. In additional experiments, we show that the learning occurs even if there are more partitions than 4, suggesting that the learning is relatively robust to the frequency of these switches. After experimenting with different learning rates λ in **LeCaR** for a variety of different workloads, we settled on a fixed learning rate of 0.45 for all the experiments.

Does LeCaR adapt to phase changes? In real data sets, *phase changes* are a frequent occurrence, where all or part of the working set gets replaced. Our next set of experiments study adaptiveness of the algorithms to different levels of phase change. As above, we show only one sample plot that explains the general behavior.

For these experiments, cache size was set at 50, with a universe of 2000 pages. Working set size was set at 30, and on each phase change 90% of the working set remained unchanged. Each phase had 5000 requests.

Figure 2 shows the results of the experiment. Again lower plot shows hit rates and upper plot reflects the inner workings of **LeCaR** with the weights of the two policies. Phase change causes a dip in the hit rate of all the algorithms. There is not much difference in the rate at which ARC and **LeCaR** recover, but **LeCaR**'s recovery rate can be regulated with the learning rate (data not shown).

The "road tests" with LeCaR. We used multi-day, block-level, production storage I/O **FIU Traces** [26] for our experiments. They include traces from an email server (mail workload), a virtual machine running two web servers (web-vm workload), and several user machines at the School of Computing and Information Sciences at FIU, collected for a duration of three weeks. The data sets labeled `casa`, `ikki`, `madmax`, `topgun` are workloads from home directories; data set `online` is from the server hosting the departments online course management system; `webresearch` is a document store for research projects; `webusers` is the home-pages of faculty/staff/students; and `mail` is the department's mail server.

We discuss our findings with Day 3 (a Monday) of each trace first. **LeCaR** outperforms ARC by as much as 18x in hit rate across the 8 production workloads from FIU [26] when caches are 0.1% of the workload size and by -4% to 30.5% when caches are 1% of the workload size. For larger caches **LeCaR** performs competitively relative to ARC (within 0.33%). Figure 3 represents

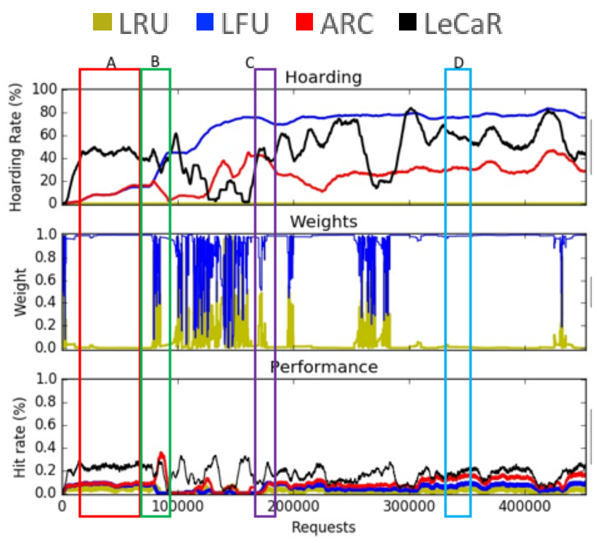

Figure 4: Using hoarding rates to explain performance for madmax workload (day 3).

only 4 of the 8 traces; the remaining four were omitted due to lack of space. Two observations are obvious. When cache sizes are high (1%) relative to the size of the workload, then all the top performers including **LeCaR** perform more or less on par. When cache sizes are low ($\leq 0.5\%$), **LeCaR** outperforms all the other competitors.

Similar characteristics were observed when we ran longer experiments with data sets representing days 1 through 7 from the collection mentioned above (data not shown). For most real data sets saturation of the hit rate seems to happen somewhere between cache sizes of 1% to 5% of the workload sizes.

Hoarding for better performance? In order to better understand the behavior of **LeCaR**, we introduce the concept of *hoarding* rate. It is defined as the percentage of entries in cache that have been accessed at least twice in the past since entering the cache, but is not among the last $2N$ unique pages accessed. By definition, LRU has zero hoarding rate because every page in the cache (including those with frequency ≥ 2) is among the last N pages accessed. LFU tends to have high hoarding rates because it cannot discard items with higher frequency as long as there is at least one lower frequency entry. LFU hoarding rate does not often decrease, except when hoarded pages are requested. Algorithms like ARC and **LeCaR** do selective hoarding, hedging their bets that some of these pages are likely to be accessed in the future. Our goal was to explain algorithm behavior by observing their hoarding behavior. See Fig. 4; here hoarding curves are smoothed by averaging over a sliding window to avoid distractions of frequent fluctuations.

We discuss regions labeled *A*, *B*, *C*, and *D* in Figure 4. In *A*, **LeCaR** is hoarding more than the other algorithms, but it is a relatively stable period where the hoarding is paying off in terms of higher hit rates. In *B*, LFU gets penalized because of prior poor choices (reflected by lowering of its weight), and **LeCaR** reacts by applying more recency criteria, thus getting rid of much of its hoarded pages. After an initial dip in hit rate, it recovers and tracks the performance of ARC, which uses its own mechanisms to react to the situation in *B*, possibly by evicting items from its high frequency queue (T_2). In *C*, some (frequent) pages are being requested after a long time, reflected by higher hit rate for LFU and a dip in its hoarding (as with **LeCaR** and ARC). However, the increase in weight for LFU pays off handsomely for **LeCaR**, which sees the highest increase in performance over its competition. *D* is similar to *A* in terms of the stability of the weights, except that the higher hoarding rates of all the algorithms is reflected in more similar hit rates.

5 Discussion and Conclusions

Large caches do not benefit from strong replacement strategies since working sets are already in cache; all good strategies perform roughly equally with insignificant differences. When cache sizes are small, subtleties of the replacement algorithms are observable. **LeCaR** relies strongly on frequency, which is important to effective cache replacement. However it tempers its reliance by using randomization and recency to clean up stale but frequent items in the cache.

LeCaR manages two data structures of metadata for each cache entry, i.e., recency and frequency. A naive implementation of **LeCaR** will incur a space overhead of $3x$ over ARC. A more careful implementation will reduce this overhead to $2x$. Futhermore, providing $2x$ additional metadata space does not improve ARC performance – on the contrary, for small cache sizes, doing so unexpectedly hurts ARC's performance. In conclusion, the reinforcement online algorithm with regret minimization when applied in a novel way to pure LRU and LFU policies results in high performance cache replacement. **LeCaR** boasts up to $18x$ improvement over the top competitor ARC for production storage I/O traces when caches are (1/1000)th of the workload size. Interestingly, the gap between **LeCaR** and ARC widens when size of the cache (relative to workload) decreases, suggesting that **LeCaR** is scalable to much larger workloads. We proposed hoarding rate as a means to understand the relative behavioral properties of these caching algorithms and to generate new insights into cache replacement analysis. The design of **LeCaR** is minimalist and uses only two policies – the vanilla LRU and LFU (without decay). If further improvements beyond

LeCaR are to be achieved, other orthogonal measures must be identified.

6 Acknowledgments

We thanks the anonymous reviewers and our shepherd, Ioan Stefanovici, for their insightful feedback. This work is supported in part by NSF awards, CCF-1718335, CNS-1563883 and CNS-1562837, and a NetApp Faculty Fellowship.

References

[1] ALI, W., SULAIMAN, S., AND AHMAD, N. Performance improvement of least-recently-used policy in web proxy cache replacement using supervised machine learning. *International Journal of Advances in Soft Computing & Its Applications 6*, 1 (2014).

[2] ARI, I., AMER, A., GRAMACY, R. B., MILLER, E. L., BRANDT, S. A., AND LONG, D. D. Acme: Adaptive caching using multiple experts. In *WDAS* (2002), pp. 143–158.

[3] BANSAL, S., AND MODHA, D. S. CAR: CLOCK with adaptive replacement. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2004), FAST '04, USENIX Association, pp. 187–200.

[4] BLASCO, P., AND GÜNDÜZ, D. Learning-based optimization of cache content in a small cell base station. In *Communications (ICC), 2014 IEEE International Conference on* (2014), IEEE, pp. 1897–1903.

[5] BLUM, A., AND MANSOUR, Y. From external to internal regret. *Journal of Machine Learning Research 8*, Jun (2007), 1307–1324.

[6] BUBECK, S., CESA-BIANCHI, N., ET AL. Regret analysis of stochastic and nonstochastic multi-armed bandit problems. *Foundations and Trends® in Machine Learning 5*, 1 (2012), 1–122.

[7] CHEN, Y., LI, Z.-Z., AND WANG, Z.-W. A ga-based cache replacement policy. In *Machine Learning and Cybernetics, 2004. Proceedings of 2004 International Conference on* (2004), vol. 1, IEEE, pp. 263–266.

[8] CONSUEGRA, M. E., MARTINEZ, W. A., NARASIMHAN, G., RANGASWAMI, R., SHAO, L., AND VIETRI, G. Analyzing adaptive cache replacement strategies. *arXiv preprint arXiv:1503.07624v2* (2017).

[9] CORBATO, F. J. A paging experiment with the MULTICS system. Tech. rep., DTIC Document, 1968.

[10] FLAXMAN, A. D., KALAI, A. T., AND MCMAHAN, H. B. Online convex optimization in the bandit setting: gradient descent without a gradient. In *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms* (2005), Society for Industrial and Applied Mathematics, pp. 385–394.

[11] FOSTER, D. J., RAKHLIN, A., AND SRIDHARAN, K. Adaptive online learning. In *Advances in Neural Information Processing Systems* (2015), pp. 3375–3383.

[12] GRAMACY, R. B., WARMUTH, M. K., BRANDT, S. A., AND ARI, I. Adaptive caching by refetching. In *Advances in Neural Information Processing Systems* (2003), pp. 1489–1496.

[13] JALEEL, A., THEOBALD, K. B., STEELY, JR., S. C., AND EMER, J. High performance cache replacement using re-reference interval prediction (rrip). *SIGARCH Comput. Archit. News 38*, 3 (June 2010), 60–71.

[14] JALEEL, A., THEOBALD, K. B., STEELY, JR., S. C., AND EMER, J. High performance cache replacement using re-reference interval prediction (rrip). In *Proceedings of the 37th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2010), ISCA '10, ACM, pp. 60–71.

[15] JANAPSATYA, A., IGNJATOVIC, A., PEDDERSEN, J., AND PARAMESWARAN, S. Dueling CLOCK: adaptive cache replacement policy based on the CLOCK algorithm. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2010* (2010), IEEE, pp. 920–925.

[16] JIANG, S., CHEN, F., AND ZHANG, X. CLOCK-Pro: An effective improvement of the CLOCK replacement. In *USENIX Annual Technical Conference, General Track* (2005), pp. 323–336.

[17] JIANG, S., AND ZHANG, X. LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Proc. ACM Sigmetrics Conf.* (2002), pp. 297–306.

[18] JIMÉNEZ, D. A., AND LIN, C. Neural methods for dynamic branch prediction. *ACM Transactions on Computer Systems (TOCS) 20*, 4 (2002), 369–397.

[19] JOHNSON, T., AND SHASHA, D. 2Q: A low overhead high performance buffer management replacement algorithm. In *Proc. of VLDB* (1994), pp. 297–306.

[20] KOLLER, R., VERMA, A., AND RANGASWAMI, R. Generalized ERSS Tree Model: Revisiting Working Sets. In *Proc. of IFIP Performance* (November 2010).

[21] KOLLER, R., VERMA, A., AND RANGASWAMI, R. Estimating Application Cache Requirement for Provisioning Caches in Virtualized Systems. In *Proc. of IEEE MASCOTS* (July 2011).

[22] LEE, D., CHOI, J., KIM, J. H., NOH, S. H., MIN, S. L., CHO, Y., AND KIM, C. S. LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE Trans. Comput. 50*, 12 (Dec. 2001), 1352–1361.

[23] LITTLESTONE, N., AND WARMUTH, M. K. The weighted majority algorithm. *Information and computation 108*, 2 (1994), 212–261.

[24] MEGIDDO, N., AND MODHA, D. S. ARC: A self-tuning, low overhead replacement cache. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2003), FAST '03, USENIX Association, pp. 115–130.

[25] O'NEIL, E. J., O'NEIL, P. E., AND WEIKUM, G. The LRU-K page replacement algorithm for database disk buffering. *SIGMOD Rec. 22*, 2 (June 1993), 297–306.

[26] R. KOLLER, R. R. I/o deduplication: Utilizing content similarity to improve i/o performance. In *Proc. 8th USENIX Conference on File and Storage Technologies* (2009), FAST 10.

[27] RAKHLIN, A., SRIDHARAN, K., AND TEWARI, A. Online learning: Beyond regret. In *Proceedings of the 24th Annual Conference on Learning Theory* (2011), pp. 559–594.

[28] ROMANO, S., AND ELAARAG, H. A neural network proxy cache replacement strategy and its implementation in the squid proxy server. *Neural computing and Applications 20*, 1 (2011), 59–78.

[29] SANTANA, R., LYONS, S., KOLLER, R., RANGASWAMI, R., AND LIU, J. To arc or not to arc. In *HotStorage* (2015).

[30] SULAIMAN, S., SHAMSUDDIN, S. M., ABRAHAM, A., AND SULAIMAN, S. Intelligent web caching using machine learning methods. *Neural Network World 21*, 5 (2011), 429.

[31] ZINKEVICH, M. Online convex programming and generalized infinitesimal gradient ascent. In *Proceedings of the 20th International Conference on Machine Learning (ICML-03)* (2003), pp. 928–936.