LFO2: An Enhanced Version of Learning-From-OPT Caching Algorithm

Yipkei Kwok and David L. Sullivan

Department of Computer Science, Mathematics and Physics,
Missouri Western State University, Saint Joseph, Missouri, USA

Abstract

Recent machine learning-based caching algorithm have shown promise. Among them, Learning-From-OPT (LFO) is the state-of-the-art supervised learning caching algorithm. LFO has a parameter named Window Size, which defines how often the algorithm generates a new machine-learning model. While using a small window size allows the algorithm to be more adaptive to changes in request behaviors, experimenting with LFO revealed that the performance of LFO suffers dramatically with small window sizes. This paper proposes LFO2, an improved LFO algorithm, which achieves high object hit ratios (OHR) with small window sizes. This results show a 9% OHR increase with LFO2. As the next step, the machine-learning parameters will be investigated for tuning opportunities to further enhance performance.

Keywords

Content Delivery Network, Cache, Machine Learning, Supervised learning

1. Introduction

When accessing Internet objects such as web pages and videos, end users experience access latency. To reduce the access latency, rather than delivering objects to their distant end users, it is increasingly common for content providers (e.g., web sites and video streaming service providers) to deliver their objects through content delivery networks [1, 2], where a content delivery network (CDN) is a geographically distributed network of cache servers. When requesting an object, an end user sends a request to a nearby cache server. If the object is available on the server, the server delivers the object directly to the end user. Because of the geographical proximity, the access latency is reduced. Otherwise, the server forwards the request to the content provider and waits for the requested object to arrive from the provider. Upon the arrival, the cache server (1) forwards the object to the end user and (2) optionally cache the object into its local storage for future accesses. While a server may blindly cache each object that it receives from content providers, admission decisions can be made by a *admission algorithm* [1]. When caching a new object, if the cache is already full, the server picks an object to evict. An *eviction algorithm* [3] decides which object(s) to evict. Therefore, the factors that decide the cache content (i.e., which objects are being cached) are (1) requests generated by end-users, (2) the admission algorithm, (3) the eviction algorithm, and (4) the cache size. Ideally, we would want as many requested objects being serviced by a cache server as is possible. The effectiveness of a cache server is quantified by its *object hit rate* (OHR) [4]. If, on average, 30 requests out of 50 are serviced directly at the cache, the OHR is 60%.

This paper addresses admission and eviction algorithms, collectively, as caching algorithms. There are two categories of caching algorithms: heuristics-based algorithms and adaption-based algorithms. Heuristics-based algorithms, such as GDSF [5], make decisions based on specific object properties, or combinations of them. Adaption-based algorithms constantly monitors request behaviors and self-adjust accordingly. This algorithm category consists of three sub-catgories: machine learning-based algorithms (e.g., LRB [6]), and non-machine learning-based algorithms (e.g., AdaptSize [1]). Existing caching algorithms will be discussed more thoroughly in the later part of this paper. Among machine learning-based techniques, LFO [7] demonstrates performance that exceeds several latest (representative) techniques in both categories of heuristics-based and adaptive-based techniques. Nonetheless, when experimenting with LFO, the experiment results revealed an opportunity to modify the algorithm, which led to significant improvement in its performance.

The structure of this paper consists of Section 2 that discusses the related work in the literature, Section 3 that provides an overview of LFO, Section 4 that discusses the proposed modification, Section 5 that illustrates the evaluations of LFO2 and results, and Section 6 that concludes this paper.

2. Related Work

Existing techniques for achieving high OHRs on CDN servers can be divided into two categories: heuristic-based technique and adapation-based techniques.

Heuristic-based techniques. These techniques make caching decisions based on assumptions related to object access properties. Examples of these properties are recency and frequency. Well-known recency-based techniques are LAMA [8], 2Q [9], and LRU-K [10], where LFU-K [11] is a well-known frequency-based technique. There are more sophisticated heuristics-based techniques that consider both recency and frequency. An example of these techniques is LHD [12]. These techniques impose a weight, either as a user-defined constant or as an adaptive variable, to decide which of the two properties is more decisive in making the caching decisions. Experiment results show that these techniques are capable of reaching the OHR of nearly 60%. While being relatively easy to implement, a drawback of heuristic-based technique is that, when their underlying assumption no longer holds, their performance suffers. Therefore, their performance may not be robust for request sequences that have varying request behaviors over time.

Adaptation-based techniques. This category of techniques continuously monitor the behavior of the request sequence and the system, and adjust their internal parameters accordingly. Among them, there are two sub-categories: machine learning-based techniques and non-machine learning-based techniques. Some machine-learning based techniques, such as Harvesting Randomness [13], are based on reinforcement learning, while some, such as LFO [7] and Pecc [14], are based on supervised learning. LFO is the state-of-the-art supervised-learning caching algorithm. Some non-machine learning-based techniques are based on Hill Climbing [15, 1, 16], where some are based on mathematical prediction [17, 18, 19]. A hill-climbing algorithm usually runs multiple simulations simultaneously, where each simulates the effect of a parameter candidate. After the simulation, the algorithm re-configures itself to adopt the best parameter candidate.

3. LFO Overview

LFO is an iterative algorithm, where *window size*, a user-specified constant, specifies the length of an iteration. The default value of window size is 1 million, which implies that a new window starts after each sequence of 1 million requests. Figure 1 describes the mechanism of LFO on a high level. During each window, for each incoming request, LFO collects information of (1) timestamp, (2) object ID, (3) object size, and (4) remaining caching space at the request arrival. These information constitutes the features of the request. At the end of each window, LFO deduces a machine-learning model to guide caching decisions during the next window. For each request, the model takes its features and decides if to cache the requested object. It is worth noting that if the model has decided to not cache an object that has just generated a hit, LFO evicts the object anyway. As a result, LFO may consume only a fraction of the cache space. LFO uses lightweight boosted decision trees based on the LGBM library [20] to create such a model using the input of (1) the features of each request arrived during the window, and (2) the best decision that it could have made for each of those requests. To deduce the latter, LFO first budget the cache resources in a window in terms of cache size (i.e., space) and window size (i.e., time). Then, LFO attempts to fit as many requests into the budget as is possible, where requests that fit into the budget are marked as *to-cache*. To maximize OHR, LFO prioritizes requests that consume little resource in terms of cache space and time, where time refers to how long the requested object stayed in the cache before generating a hit. LFO marks requests that do not fit into the budget as *not-to-cache*. This paper calls the resultant sequence of decisions the *optimal caching decisions* and the process described above that produces the sequence *Optimal Caching*. Practically, such a model maps the feature of each request to the best decision that could have been made. In the next window, given the features of each request, the model predicts the best caching decision for the request aiming at maximizing the OHR of the next window.

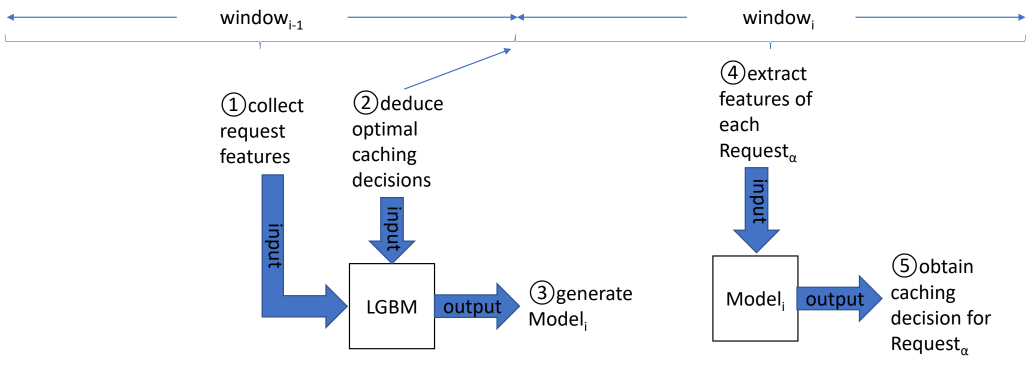
window$_{i-1}$
window$_i$
①collect request features
②deduce optimal caching decisions
input
input
LGBM
output
③generate Model$_i$
④extract features of each Request$_\alpha$
input
Model$_i$
output
⑤obtain caching decision for Request$_\alpha$

Figure 1: Using the request features collected throughout a window, LFO deduces the optimal caching decisions and generates a model for making caching decisions in the next window. [7]

4. The Effects of Window Size on Performance and LFO2

When experimenting with LFO, experiment results revealed that Window Size has a significant impact on the performance of LFO in terms of OHR. Section 4.1 presents the performance results of LFO with different window sizes and the analysis of the performance differences. While the results show that LFO performs better with large window sizes, using small window sizes allows LFO to be more adaptive to changes in request behaviors. This observation agrees with the common understanding in the literature [21, 22, 23]. Based on this performance analysis, This paper proposes LFO2, a revised LFO design, which allows LFO to achieve high OHRs using small window sizes. Section 4.2 presents LFO2 in detail.

4.1 Window Size Influences on The Performance

The default window size of LFO is 1 million. LFO, which was implemented on WebCacheSim2 [24], was evaluated with window sizes of 1, 2, 4, and 8 millions using the CDN1 trace [25]. Table 1 summarizes the results. As Window Size increases, OHR increases. Achieving high OHRs requires simultaneously meeting two necessary conditions. First, optimal caching must produce a sequence of caching decisions that leads to high OHR. Second, the machine-learning models produce caching decisions that closely assimilate the optimal caching decisions. To quantify the quality of caching decisions produced by the models, this paper introduces a metric named *Decision Accuracy*, which measures the percentages of caching decisions generated by the models that match the optimal caching decisions.

Table 1: Effects of Window Size on OHR, Decision Accuracy, and the number of False Negatives.

window size (million)	OHR	decision accuracy (%)	false negatives (%)
1	48.72%	84.12%	39.96%
2	58.08%	82.22%	33.49%
4	67.29%	81.60%	26.85%
8	74.76%	82.75%	21.35%

As shown in Table 1, the general trend is that decision accuracy decreases as window size increases with the exception of the window size of 8M requests, which slightly exceeds the decision accuracy when the window size of 2M requests by approximately 0.5%. As window size is doubled, the amount of data used for training is doubled accordingly, which enhances the accuracy of the model [26]. This suggests why the 8 million-requests window size achieves a decision accuracy higher than when the window sizes are 2 millions and 4 millions requests.

However, it is commonly known that request behaviors, including reuse distance and object size, vary over time [21, 22, 27]. As a result, a model generated at the beginning of a window may no longer accurately reflect the request behaviors during the later part in the window. Therefore, this explains why the decision accuracy decreases as the window size increases. Despite of the 8 million-request window size being an exception, the window size of 1 million request still yields the highest decision accuracy among all window sizes.

While the decision accuracy values suggest that our models are fairly accurate, OHRs remained low with small window sizes. Given the two necessary conditions for achieving high OHRs discussed above, the only plausible explanation is that Optimal Caching is unable to produce caching decisions that lead to high OHRs when windows are small. An in-depth analysis shows that Optimal Caching misses many caching opportunities, where it should mark the requests as to-cache as those requests may yield hits with relatively little resources, in terms of space and time. These missed opportunities happen at the final request to each object that is accessed during a window. In the absence of future access information, Optimal Caching assumes that the requested object will not be accessed again and marks request as not-to-cache. While reasonable, those requests may potentially lead to hits if they were, otherwise, marked as to-cache. This paper calls those requests *false negatives*. Figure 2 illustrate how reducing the window size increases the number of false negatives. Given the full knowledge of object access in the trace, we would ideally mark each of the first 8 requests, except the second request to Object 3, as to-cache. Given the window size of 8 requests, without knowing that Objects 1, 2, and 4 will soon be accessed at the beginning of the next window, Optimal Caching marks the last 3 requests in the window as not-to-cache. However, marking them as to-cache would yields 3 additional hits in the next window. Therefore, the last 3 requests in the window are false negatives. As shown on the lower part of the figure, as the window size is reduced by half to 4, the number of false negatives increases to 6. The smaller the window size, the more false negatives there are in the optimal caching decisions. This increases the likeliness that the machine-learning model decides to not cache the accessed objects. This explains why OHR increases as the window size increases.

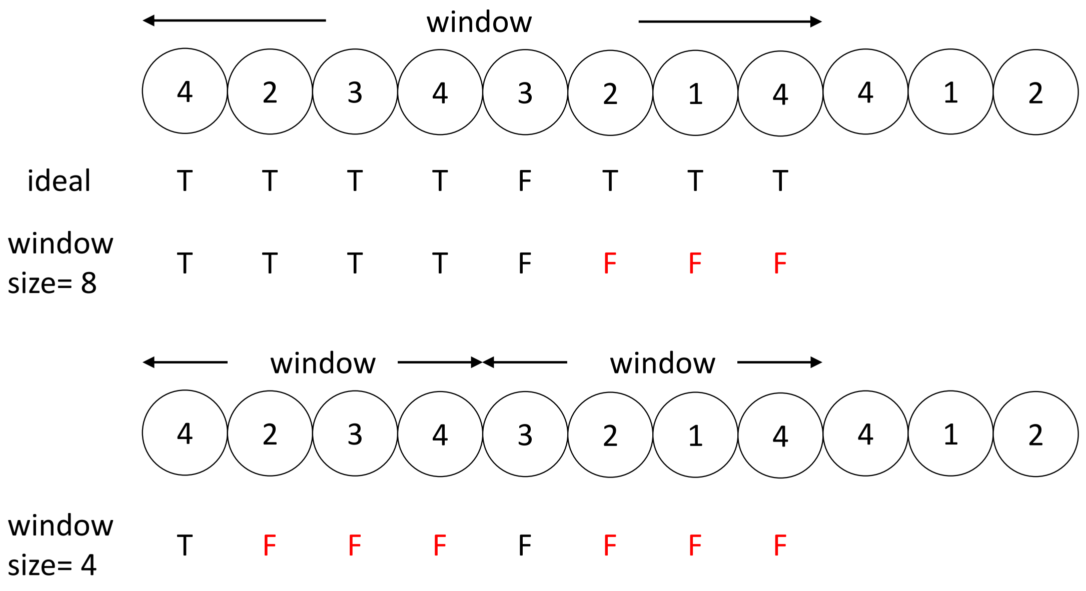

Figure 2: The smaller the window size, the more request are marked as not-to-cache. F's denote false negatives.

To count the number of false negatives in the window sizes of 1, 2, 4, and 8 millions, an ideal sequence of caching decisions, as shown in Figure 2, is needed. Since the number of false negatives decreases as the window size increases, the window size that equals to the trace length yields the sequence of caching decisions containing the fewest false negatives possible. This is because the caching decisions are made with the full knowledge of object accesses. This paper calls such sequence the *ultimate caching decisions*, where the process for deducing the sequence *Ultimate Caching*.

The sequences of optimal and ultimate caching decisions were compared to obtain the percentage of false negatives for each window size, which is presented in Table 1. By increasing the window size from 1 million to 8 millions, OHR increases by over 25% while the percentage of false negatives drops from 19%. This observation hints us that if we manage to reduce false negatives by a certain amount with the window size of 1 million, we should be increase the OHR by a similar amount.

4.2 LFO2 Algorithm

This is how LFO2 reduces the number of false negatives. When its Optimal Caching encounters the final request to each object accessed during a window, rather than marking it as not-to-cache, it gives the request a ``second chance," by consulting the model for the likelihood that the requested object will be accessed again. If the likelihood is high enough, LFO2 marks it as to-cache. Since this is the second chance, it is reasonable for the request to be subject to higher requirement, in terms of the likelihood. LFO2 marks the request as to-cache only if its likelihood is higher than a threshold, known as *Likelihood Threshold* in this paper. The idea of giving objects that are about to be evicted a second chance has been shown to be enhance performance in areas such as memory paging [28]. Note that it is inevitable that LFO2 mistakenly marks a request as to-cache when trying to give the request a second chance. In the actual request arrival sequence, those requests may actually consume much cache resource in order to generate hits and, therefore, should not be cached. This paper calls those requests *false positives*. As for false negatives, false positives are identified by comparing the sequences of ultimate and optimal caching decisions. Section 5 presents the performance results of LFO2 and the effects of Likelihood threshold on the numbers of false negatives and positives.

5. Evaluation

This section evaluates the effectiveness of LFO2 and compares its performance with that of LFO. Section 5.1 discusses the experiment setup that we used for the evaluation, while Section 5.2 discusses and analyzes the evaluation results.

5.1 Experiment Methodology

We implemented LFO2 on WebCacheSim2 and evaluated it with the CDN1 trace that we used in Section 4.1. To ensure that none of the caching algorithm may achieve high OHRs by naively caching every object requested for future use, the cache size was set to 256GB, which is only one tenth of the total size of objects requested by the cache. As mentioned in Section 4, LFO attempts to correct the false negatives among the optimal caching decisions. To quantify the optimality of the optimal caching decisions generated by LFO2, we measured the percentage of optimal caching decisions that matched the ultimate caching decisions. Among the mismatches, there are false negatives and false positives, where the latter are those requests that Ultimate Caching recommends not to cache while LFO2 mistakenly suggested to cache. To understand how LFO2 affect the cache space consumption, we reported the maximum amount of cache space consumed in each experiment. The literature recognizes that small windows all scheduling algorithms to be more adaptive to changes in workload characteristics through more frequent self-adjusting. To reaffirm this common belief, we measured the *Decision Correctness*, the percentage of decisions made by the model during a window that matches the optimal caching decisions deduced at the end of the window.

5.2 Evaluation Results

To evaluated the claim that LFO2 is capable of achieving high OHRs even with small windows, LFO2 was experimented with the window size of 2 million requests. Table 2 presented the results. The table also includes the LFO results for window sizes of 2 and 8 millions for reference. Note that, among the 4 window sizes experimented with LFO, it performed the best with the window size of 8 millions. LFO2 outperformed LFO with the window size of 8 million requests by nearly 9%. It achieved so by dramatically reducing the number of false negatives, by up to 33%, as shown in the table. The results show that, as Likelihood Threshold decreases, the number of false negatives decrease accordingly. This is because, the lower the threshold is, the more likely Optimal Caching marks the final request to each object begin accessed during a window as to-cache.

Table 2: The performance (OHR) of LFO2 with window size of 2 million requests. Data of false negatives and positives and maximum cache occupancy are presented to help understanding performance. LFO results of window sizes of 2 and 8 million requests are presented for readers' reference.

	window size (million)	Likelihood threshold	OHR	false negatives (%)	false positives (%)	maximum cache occupancy (GB)
LFO	2	N/A	58.08%	33.48%	0.00%	58
LFO	8	N/A	74.76%	74.76%	0.00%	177
LFO2	2	0.5	83.64%	0.53%	3.73%	254
	2	0.6	83.65%	0.74%	3.72%	254
	2	0.7	83.31%	1.01%	3.70%	254
	2	0.8	81.42%	4.60%	3.49%	254
	2	0.9	58.10%	33.44%	0.00%	58

However, lowering the likelihood threshold also increases the possibility that Optimal caching mistakenly marks requests as to-cache. This explains why, as Likelihood Threshold decreases, the percentage of false positives increased, though slightly. Nonetheless, the percentage of false positives never exceeded 4%.

By reducing false negatives in optimal caching decisions, LFO2 creates models that have higher tendency to cache objects. This is reflected at the max cache occupancy (i.e., the largest amount of cache occupied throughout an experiment). For LFO, as the window size increases from 2 to 8 million requests, the the models become more inclined to cache objects. As a result, the maximum cache occupancy increases from 58 to 177GB. Similarly, in LFO2, with Likelihood Threshold below 0.9, the models identified more requests, whose accessed objects, if cached, would generate hits to enhance the OHR. Therefore, the maximum cache occupancy further increases to 254GB, which almost filled up the cache. This paper argues that the increase in occupancy is justifiable based on the OHR improvements by LFO2. Indeed, in the literature, high resource utilization is a desirable property.

6. Conclusion

Achieving high OHRs on content delivery networks is crucial to end-user experience when accessing Internet content. Among machine learning-based techniques for achieving high OHR, LFO is one of the state-of-the-art algorithms. This paper presented LFO2, an improved version of LFO. LFO2's machine-learning models are more accurate in identifying objects that increase the OHR, if cached. We evaluated LFO2 using a trace of requests captured in a production environment. LFO2 achieves a 9% OHR improvement over LFO. While LFO2 almost fully occupied the entire cache in order to achieve such a performance improvement, we see it as LFO2's advantage as it better utilizes cache resources. In the future, the possibility of machine-learning parameter tuning for enhancing caching performance will be investigated.

References

[1] D. S. Berger, R. K. Sitaraman, and M. Harchol-Balter, ``Adaptsize: Orchestrating the hot object memory cache in a content delivery network," in *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, 2017, pp. 483--498. *Cited on page(s):* 1, 2

[2] A. Blankstein, S. Sen, and M. J. Freedman, ``Hyperbolic caching: Flexible caching for web applications," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, 2017, pp. 499--511. [Online]. Available: https://www.usenix.org/conference/atc17/technical-sessions/presentation/blankstein *Cited on page(s):* 1

[3] M. Bilal and S.-G. Kang, ``A cache management scheme for efficient content eviction and replication in cache networks," *IEEE Access*, vol. 5, pp. 1692--1701, 2017. *Cited on page(s):* 1

[4] D. Menasce, ``Scaling web sites through caching,'' *IEEE Internet Computing*, vol. 7, no. 4, pp. 86--89, 2003. *Cited on page(s):* 1

[5] Z. Zhao, Y. Ma, and Q. Cong, ``Gdsf-based low access latency web proxy caching replacement algorithm,'' in *Proceedings of the 2018 2nd International Conference on Computer Science and Artificial Intelligence*, ser. CSAI '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 232–236. [Online]. Available: https://doi.org/10.1145/3297156.3297237 *Cited on page(s):* 1

[6] Z. Song, D. S. Berger, K. Li, and W. Lloyd, ``Learning relaxed belady for content distribution network caching,'' in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 529--544. [Online]. Available: https://www.usenix.org/conference/nsdi20/presentation/song *Cited on page(s):* 1

[7] D. S. Berger, ``Towards lightweight and robust machine learning for cdn caching,'' in *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*, 2018, pp. 134--140. *Cited on page(s):* 2, 3

[8] X. Hu, X. Wang, Y. Li, L. Zhou, Y. Luo, C. Ding, S. Jiang, and Z. Wang, ``LAMA: Optimized locality-aware memory allocation for key-value cache,'' in *2015 {USENIX} Annual Technical Conference ({USENIX}{ATC} 15)*, 2015, pp. 57--69. *Cited on page(s):* 2

[9] T. Johnson, D. Shasha *et al.*, ``2q: a low overhead high performance bu er management replacement algorithm,'' in *Proceedings of the 20th International Conference on Very Large Data Bases*. Citeseer, 1994, pp. 439--450. *Cited on page(s):* 2

[10] E. J. O'neil, P. E. O'neil, and G. Weikum, ``The lru-k page replacement algorithm for database disk buffering,'' *Acm Sigmod Record*, vol. 22, no. 2, pp. 297--306, 1993. *Cited on page(s):* 2

[11] L. B. Sokolinsky, ``Lfu-k: An effective buffer management replacement algorithm,'' in *Database Systems for Advanced Applications*, Y. Lee, J. Li, K.-Y. Whang, and D. Lee, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 670--681. *Cited on page(s):* 2

[12] N. Beckmann, H. Chen, and A. Cidon, ``LHD: Improving cache hit rate by maximizing hit density,'' in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, Apr. 2018, pp. 389--403. [Online]. Available: https://www.usenix.org/conference/nsdi18/presentation/beckmann *Cited on page(s):* 2

[13] M. Lecuyer, J. Lockerman, L. Nelson, S. Sen, A. Sharma, and A. Slivkins, ``Harvesting randomness to optimize distributed systems,'' in *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, 2017, pp. 178--184. *Cited on page(s):* 2

[14] A. Bhardwaj and V. Janardhan, ``Pecc: Prediction-error correcting cache,'' in *Workshop on ML for Systems at NeurIPS*, vol. 32, 2018. *Cited on page(s):* 2

[15] S. Bansal and D. S. Modha, ``Car: Clock with adaptive replacement.'' in *FAST*, vol. 4, 2004, pp. 187--200. *Cited on page(s):* 2

[16] A. Cidon, A. Eisenman, M. Alizadeh, and S. Katti, ``Cliffhanger: Scaling performance cliffs in web memory caches,'' in *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*, 2016, pp. 379--392. *Cited on page(s):* 2

[17] G. Almási, C. Caşcaval, and D. A. Padua, ``Calculating stack distances efficiently,'' in *Proceedings of the 2002 workshop on Memory system performance*, 2002, pp. 37--43. *Cited on page(s):* 2

[18] X. Hu, X. Wang, L. Zhou, Y. Luo, Z. Wang, C. Ding, and C. Ye, ``Fast miss ratio curve modeling for storage cache,'' *ACM Transactions on Storage (TOS)*, vol. 14, no. 2, pp. 1--34, 2018. *Cited on page(s):* 2

[19] C. A. Waldspurger, N. Park, A. Garthwaite, and I. Ahmad, ``Efficient {MRC} construction with {SHARDS},'' in *13th {USENIX} Conference on File and Storage Technologies ({FAST} 15)*, 2015, pp. 95--110. *Cited on page(s):* 2

[20] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu, ``Lightgbm: A highly efficient gradient boosting decision tree," *Advances in neural information processing systems*, vol. 30, pp. 3146--3154, 2017. *Cited on page(s):* 2

[21] S. Arunagiri, Y. Kwok, P. J. Teller, R. Portillo, and S. R. Seelam, ``A heuristic for selecting the cycle length for FAIRIO," Department of Computer Science, The University of Texas at El Paso, Tech. Rep. UTEP-CS-11-53, September 2011. *Cited on page(s):* 3, 4

[22] S. Arunagiri, Y. Kwok, P. Teller, R. Portillo, and S. Seelam, ``Fairio: An algorithm for differentiated i/o performance," in *Computer Architecture and High Performance Computing (SBAC-PAD), 2011 23rd International Symposium on*, oct. 2011, pp. 88 --95. *Cited on page(s):* 3, 4

[23] S. Arunagiri, Y. Kwok, P. J. Teller, R. A. Portillo, and S. R. Seelam, ``Fairio: A throughput-oriented algorithm for differentiated i/o performance," *International Journal of Parallel Programming*, vol. 42, no. 1, pp. 165--197, Feb 2014. [Online]. Available: http://dx.doi.org/10.1007/s10766-012-0217-6 *Cited on page(s):* 3

[24] ``Webcachesim2 - a simulator for cdn caching and web caching policies." https://github.com/sunnyszy/lrb, accessed: 2021-07-30. *Cited on page(s):* 3

[25] ``Data: publicly available cdn request trace (from a cdn server in san francisco serving wikipedia production traffic) used in the evaluation." https://www.cs.cmu.edu/~dberger1/data_and_software.html, accessed: 2021-07-30. *Cited on page(s):* 3

[26] ``Lgbm parameter tuning for better accuracy," https://lightgbm.readthedocs.io/en/latest/Parameters-Tuning.html?highlight=parameter%20tuning, accessed: 2021-07-30. *Cited on page(s):* 3

[27] S. Arunagiri, Y. Kwok, P. Teller, R. Portillo, and S. Seelam, ``FAIRIO: a throughput-oriented algorithm for differentiated I/O performance," *Int. J. of Parallel Programming*, pp. 1--33, 2012. *Cited on page(s):* 4

[28] K. M. Saleem, M. Iqbal, H. Saadi, F. Qazi, and D.-e.-S. Agha, ``Second chance page replacement algorithm with optimal (scao)," in *2019 International Conference on Information Science and Communication Technology (ICISCT)*, 2019, pp. 1--5. *Cited on page(s):* 5