Learning Cache Replacement with CACHEUS

Liana V. Rodriguez, Farzana Yusuf, Steven Lyons, Eysler Paz, Raju Rangaswami, and Jason Liu, *Florida International University;* Ming Zhao, *Arizona State University;* Giri Narasimhan, *Florida International University*

https://www.usenix.org/conference/fast21/presentation/rodriguez

This paper is included in the Proceedings of the 19th USENIX Conference on File and Storage Technologies.

February 23–25, 2021

978-1-939133-20-5

Open access to the Proceedings of the 19th USENIX Conference on File and Storage Technologies is sponsored by USENIX.

Learning Cache Replacement with CACHEUS

Liana V. Rodriguez[†*] Farzana Yusuf[†*] Steven Lyons[†] Eysler Paz[†]
Raju Rangaswami[†] Jason Liu[†] Ming Zhao[‡] Giri Narasimhan[†]
[†] *Florida International University* [‡] *Arizona State University*

Abstract

Recent advances in machine learning open up new and attractive approaches for solving classic problems in computing systems. For storage systems, cache replacement is one such problem because of its enormous impact on performance. We classify workloads as a composition of four workload primitive types — *LFU-friendly, LRU-friendly, scan*, and *churn*. We then design and evaluate CACHEUS, a new class of fully adaptive, machine-learned caching algorithms that utilize a combination of experts designed to address these workload primitive types. The experts used by CACHEUS include the state-of-the-art ARC, LIRS and LFU, and two new ones – SR-LRU, a scan-resistant version of LRU, and CR-LFU, a churn-resistant version of LFU. We evaluate CACHEUS using 17,766 simulation experiments on a collection of 329 workloads run against 6 different cache configurations. Paired t-test analysis demonstrates that CACHEUS using the newly proposed lightweight experts, SR-LRU and CR-LFU, is the most consistently performing caching algorithm across a range of workloads and cache sizes. Furthermore, CACHEUS enables augmenting state-of-the-art algorithms (e.g., LIRS, ARC) by combining it with a complementary cache replacement algorithm (e.g., LFU) to better handle a wider variety of workload primitive types.

1 Introduction

Cache replacement algorithms have evolved over time with each algorithm attempting to address some shortcomings of previous algorithms. However, despite the many advances, state-of-the-art caching algorithms continue to leave room for improvement. First, as demonstrated abundantly in the literature, caching algorithms that do well for certain workloads do not perform well for others [23, 13, 20, 12, 29, 34]. The production storage workloads of today are significantly diverse in their characteristic features and these features can vary over time even within a single workload. Second, as demonstrated recently [34], caching algorithms that do well for certain cache sizes do not necessarily perform well for other cache sizes. Indeed, the workload-induced dynamic cache state, the cache-relevant workload features, and thereby the most effective strategies, can all vary as cache size changes.

The ML-based LeCaR algorithm demonstrated that having access to two simple policies, LRU and LFU was sufficient to outperform ARC across specific production-class workloads. LeCaR used *regret minimization* [22, 21], a machine learning technique that allowed the dynamic selection of one of these policies upon a *cache miss*. We review LeCaR both analytically and empirically to demonstrate that while LeCaR took a valuable first step, it had significant limitations. As a result, LeCaR underperforms state-of-the-art algorithms such as ARC, LIRS, and DLIRS for many production workloads.

As our first contribution, we identify the cache-relevant features that inform *workload primitive types*. In particular, we identify four workload primitive types: *LRU-friendly*, *LFU-friendly*, *scan*, and *churn*. The workload primitive types vary across workloads, within a single workload over time, and as cache size changes. Our second contribution, CACHEUS, is inspired by LeCaR but overcomes an important shortcoming by being completely *adaptive*, with the elimination of all statically chosen hyper-parameters, thus ensuring high flexibility. Our third contribution is the design of two lightweight experts, CR-LFU and SR-LRU; put together, these address a broad range of workload primitive types. CR-LFU infuses LFU with *churn resistance* and SR-LRU infuses LRU with *scan resistance*. CACHEUS when using the proposed two experts is able to perform competitively or better for a significant majority of the (workload, cache-size) combinations when compared with the state-of-the-art.

We evaluate CACHEUS using 17,766 simulation experiments on a workload collection comprising of over 329 individual single-day workloads sourced from 5 different production storage I/O datasets. For each workload, we evaluate against 6 different cache configurations that are sized relative to the individual workload's *footprint*, the set of unique data accessed. We perform *paired t-tests* analysis comparing CACHEUS against individual algorithms across 30 different (workload, cache-size) combinations. CACHEUS using SR-LRU and CR-LFU as experts is the most consistently performing algorithm with 87% of the workload-cache combinations being the best or indistinguishable from the best performing algorithm, and distinctly different than the best performing algorithm for the remaining 13%. For the 13%

[*]The first two authors contributed equally to this work.

Dataset	# Traces	Footprint	Requests	Details
FIU [33, 16]	184	398MB	314563	End user home directories; Webpage and web-based email servers; Online course management system
MSR [33, 24]	22	467MB	4126937	User home and Project directories; Hardware monitoring; Source control; Web staging; Terminal, Web/SQL, Media, Test web servers; Firewall/web proxy
CloudPhysics [35]	99	458MB	2470326	VMware VMs from cloud enterprise
CloudVPS [2]	18	3.7GB	3400025	VMs from cloud provider
CloudCache [2]	6	6.2GB	3867313	Online course website; CS department web server

Table 1: Descriptions for the 5 datasets used (average *footprint* and requests). *Each trace has a 1 day duration.*

cases where an algorithm other than CACHEUS is found to be distinctly better, no single algorithm is found to be consistently the best, indicating that CACHEUS is a good default choice. Finally, when using state-of-the-art algorithms such ARC and LFU, we show that the CACHEUS framework provides a simple way to enable access to an additional expert with complementary expertise such as LFU. These CACHEUS variants achieve at least competitive performance when compared against the original algorithms and other competitors.

2 Motivation

2.1 Understanding Workloads

Caching algorithms in the past have optimized for specific workload properties. As today's workloads continue to increase in complexity, even state-of-the-art algorithms demonstrate inconsistent performance. To understand the production storage workloads of today, we analyzed over 329 production storage traces sourced from 5 different production collections (see Table 1).

2.1.1 Workload Primitive Types

Based on our analysis of production storage workloads, we define the following set of workload primitive types.

- **LRU-friendly** defined by an access sequence that is best handled by the *least recently used (LRU)* caching algorithm.
- **LFU-friendly** defined by an access sequence that is best handled by the *least frequently used (LFU)* caching algorithm.
- **Scan** defined by an access sequence where a subset of stored items are accessed exactly once.
- **Churn** defined by repeated accesses to a subset of stored items with each item being accessed with equal probability.

Figure 1 shows an example of how the workload primitive types manifest in a production trace from the FIU collection. As one may notice, the primitive types are not all exclusive — for instance, a workload that's *LRU-friendly* may also manifest the *churn* type. Our goal was identifying workload primitive types that would directly inform specific, yet distinct, caching decisions.

We found that most of the workloads that we examined contained at least one occurrence of each of the workload primitive types. However, these workloads were not all the same in their composition. For instance, the MSR collection contains all the primitive types with one of the workloads *(proj3)* mostly comprising a single long scan. A summary of our findings are presented in Table 2.

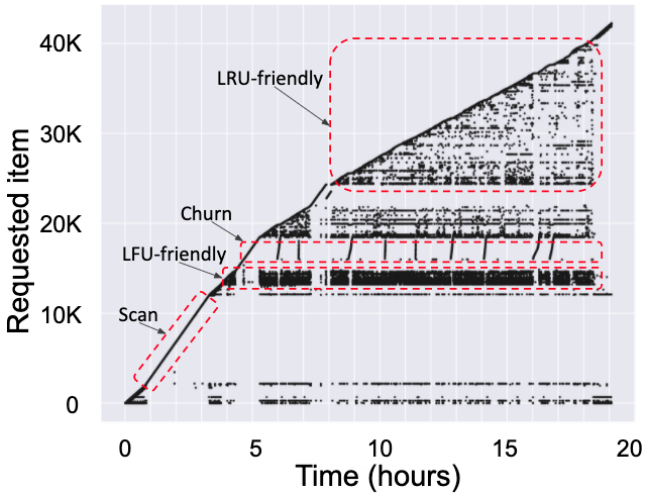

Figure 1: Access pattern for the *topgun (day* 16*)* workload from the FIU trace collection. *Dashed lines highlight manifestation of workload primitive types.*

2.1.2 Composing Workloads

Modern storage workloads are typically a composition of the above workload primitive types. Furthermore, as the cache size changes, a single workload's primitive type may vary. For instance, an *LRU-friendly* type workload at cache size C_1 may transform into a *Churn* type at a cache size $C_2 < C_1$. This can occur when items in the workload's LRU-friendly working set start getting removed from the cache prior to being reused. Figure 2 illustrates this phenomenon by comparing the performance of LRU against the churn-friendly CR-LFU algorithm proposed in this paper. Finally, storage working sets are telescoping in nature with larger subsets

Dataset	Churn	Scan	LRU	LFU
FIU [33, 16]	✓	✓	✓	✓
MSR [33, 24]	✓	✓	✓	✓
CloudPhysics [35]	✓	✓	✓	✓
CloudVPS [2]	✓	✓	✓	✓
CloudCache [2]	✓	✗	✓	✓

Table 2: Workload Primitive Types identified using algorithms that optimize for each primitive type.

Algorithm	Churn	Scan	LRU	LFU
ARC	✗	✓	✓	✗
LIRS*	✗	✓	✗	✗
LeCaR*	✓	✗	✓	✓
DLIRS	✗	✓	✓	✗

Table 3: Caching algorithms handling of workload primitive types. *Parametric algorithms are noted using an *.*

of items accessed at a lower frequency often each entirely subsuming one or more smaller subsets of items accessed at a higher frequency [17, 27]. The LeCaR [34] algorithm was the first to demonstrate an ability to adapt its behavior based on the available cache size, independent of the ability to adapt to the dynamics of the workload.

2.2 Caching Algorithms

Adaptive Replacement Cache (ARC): ARC [23] is an adaptive caching algorithm that is designed to recognize both recency and frequency of access. ARC divides the cache into two LRU lists, T_1 and T_2. T_1 holds items accessed once while T_2 keeps items accessed more than once since admission. Since ARC uses an LRU list for T_2, it is unable to capture the full frequency distribution of the workload and perform well for LFU-friendly workloads. For a *scan* workload, new items go through T_1 protecting frequent items previously inserted into T_2. However, for *churn* workloads, ARC's inability to distinguish between items that are equally important leads to continuous cache replacement [29].

Low Interference Recency Set (LIRS): LIRS [13] is a state-of-the-art caching algorithm based on reuse distance. LIRS handles *scan* workloads well by routing one-time accesses via its short filtering list Q. However, LIRS's ability to adapt is compromised because of its use of a fixed-length Q. In particular, if reuse distances exceed the 1% length, LIRS is unable to recognize reuse quickly enough for items with low overall reuse. And, similar to ARC, LIRS does not have access to the full frequency distribution of accessed items which limits its effectiveness for *LFU-friendly* workloads.

Dynamic LIRS (DLIRS): DLIRS [20] is a recently proposed caching policy that incorporates adaptation in LIRS. DLIRS dynamically adjusts the cache partitions assigned to high and low reuse-distance items. Although this strategy achieves performance comparable to ARC for some cache size configurations with *LRU-friendly* workloads

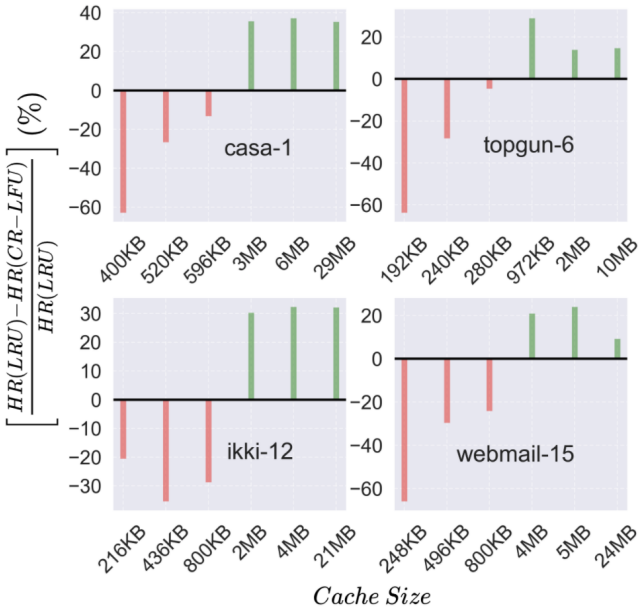

Figure 2: Relative difference in hit-rate (HR) of LRU and CR-LFU for *casa, topgun, ikki, and webmail* workloads from the FIU trace collection.

while maintaining LIRS's behavior for scans, we found its performance inconsistent across the workloads we tested against. Finally, it inherits the LFU-unfriendliness of LIRS.

Learning Cache Replacement (LeCaR): LeCaR [34] is a machine learning-based caching algorithm that uses reinforcement learning and regret minimization to control its dynamic use of two cache replacement policies, LRU and LFU. LeCaR was shown to outperform ARC for small cache sizes for real-world workloads [34]. However, LeCaR has drawbacks relating to adaptiveness, overhead, and churn-friendliness. In Section 3, we discuss these limitations further.

In Table 3, we compare the current state-of-the-art algorithms in terms of their ability to handle various workload primitive types.

2.3 Need for a New Approach

Each of the state-of-the-art caching algorithms address a subset of workload primitive types. We conducted an empirical study using over 329 storage I/O traces from 5 different production systems, across 6 different workload-specific cache configurations — from 0.05% to 10% of the workload footprint. To understand relative performance across such a large collection of experiments, we ranked algorithms based on their achieved hit-rates for individual workloads. The best-performing algorithm received the rank of 1 as well as any other algorithm that achieved a hit-rate within a 5% *relative* margin. For example, if the best-performing algorithm achieves a hit-rate of 40%, any other algorithm that achieves a hit-rate within the range 38% to 40% is also ranked as 1, but anything lower than 38% is ranked 2 or higher. Next,

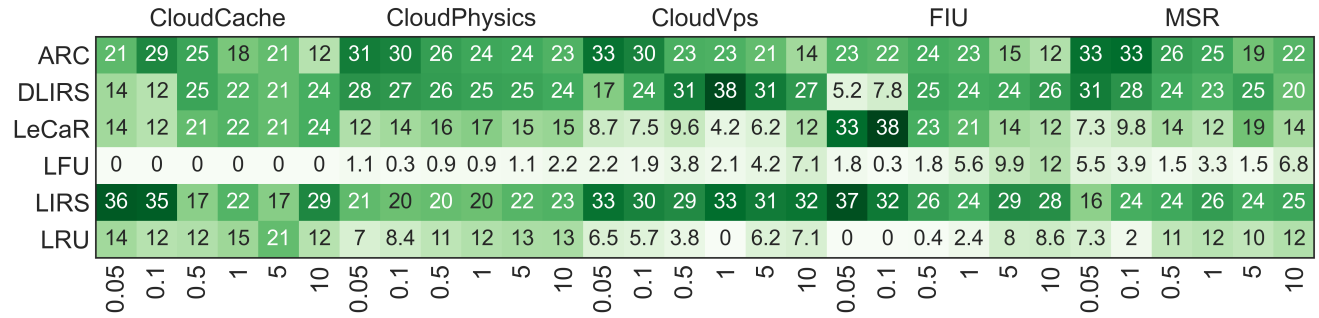

Figure 3: An analysis of the ranked performance of state-of-the-art caching algorithms. *The X-axis indicates cache size as a % of workload footprint. A darker cell indicates that an algorithm's performance across all workloads of the dataset was better. The number within each cell denotes the percentage of workloads for which an algorithm was ranked 1. For example, ARC has the highest hit-rate in 34% of the workloads for MSR at the 0.05% cache size.*

we computed the percentage of workloads within each set for which a given algorithm was assigned a rank of 1. We present this information in Figure 3.

Of the state-of-the-art caching algorithms, we observe that no algorithm is a clear winner. For instance, while LIRS achieves the best performance for CloudCache workloads at cache sizes of 0.05% and 0.1%, ARC outperforms the rest of the competitors for a majority of the MSR workloads, and LeCaR is the best for FIU workloads at a cache size of 0.1%. New caching algorithms that perform competitively across a wide range of workloads and cache configurations would be valuable.

3 CACHEUS

Given the distinct characteristics and dynamic manifestation of workload primitive types within a workload over time, caching algorithms need to be both nimble and adaptive. Online reinforcement learning is valuable because of its inherent ability to adapt to the unknown dynamics of the system being learned. CACHEUS uses online reinforcement learning with regret minimization to build a caching algorithm that attempts to optimize for dynamically manifesting workload primitive types. Since CACHEUS' design draws heavily from LeCaR, we review it briefly first, conduct an investigative study of LeCaR, and finally discuss the CACHEUS algorithm.

3.1 LeCaR: A Review

LeCaR demonstrated the feasibility of building a caching system that uses reinforcement learning and regret minimization. LeCaR learns the optimal eviction policy dynamically, choosing from exactly two basic experts, LRU and LFU. On each eviction, an expert is chosen randomly with probabilities proportional to the weights w_{LRU} and w_{LFU}. LeCaR dynamically learns these weights by assigning penalties for wrongful evictions.

To control online learning, LeCaR uses a *learning rate* parameter to set the magnitude of the change when the algorithm makes a poor decision. Larger learning rates allow quicker learning, but need larger corrections when the learning is flawed. LeCaR uses a *discount rate* parameter to decide how quickly to stop learning.

3.2 Running Diagnostics on LeCaR

In over 17,766 distinct caching simulations that we ran against LeCaR using 329 workloads, we found that experts other than LRU and LFU produced outcomes that were significantly better for a non-trivial number of workloads. In particular we found that LRU and LFU were unable to address the *scan* and *churn* workload primitive types. This motivates further exploration of the choice of experts for learning cache replacement within the regret minimization framework.

A second challenge when using LeCaR in practice is the manual configuration necessary for its two internal parameters — learning rate and discount rate. These parameters were fixed after experimenting with many workloads in LeCaR [34]. From the above empirical evaluation, we found that eliminating the discount rate altogether did not affect LeCaR's performance appreciably. Furthermore, different static values of the learning rate were found to be optimal for different workloads (see Figure 4). In addition, we observed across almost all workloads that not only do workload characteristics change substantially over time, the velocity and magnitude of these changes also varied significantly over time. To accommodate this dynamism, different values for the learning rate were found to be optimal at different points in time.

3.3 Formalizing CACHEUS(A,B)

CACHEUS starts off by simplifying and adapting LeCaR. First, for reasons discussed previously, CACHEUS simply eliminates the use of *discount rate*. Second, for adapting the *learning rate* hyper-parameter, we investigated adaptation approaches including grid search, random search [5], gaussian, bayesian and population based approaches [14, 36, 32, 6, 3, 19], and gradient-based optimization [26, 7, 15, 28, 37,

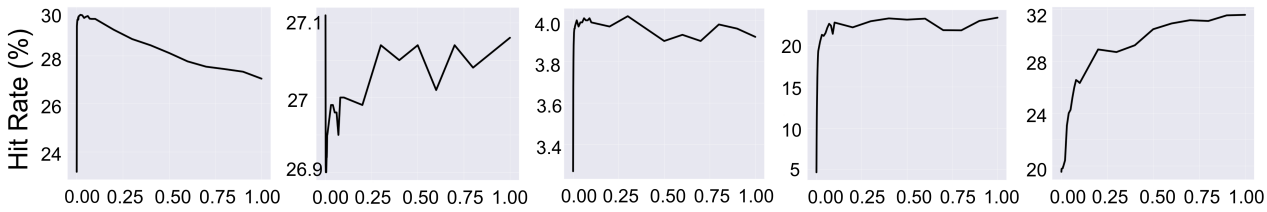

Figure 4: The optimal learning rate varies across workloads. *X-axis indicates learning rates. Cache size was chosen as 0.1% of workload footprint. We chose one workload each from CloudCache, CloudPhysics, CloudVPS, FIU, and MSR (from left to right).*

25, 8]. Ultimately, we chose a gradient-based stochastic hill climbing approach with random restart [31] for CACHEUS, a choice that proved to be the most consistent. Using this technique, at the end of every window of N requests (N = cache size), the gradient of the performance (average hit-rate) with respect to the learning rate over the previous two windows is calculated. If the gradient is positive (negative, resp.), then the direction of change of the learning rate is sustained (reversed, resp.). The amount of change of learning rate in the previous window determines the magnitude of the change in learning rate for the next window. Therefore, if the performance increases (decreases, resp.) by increasing the learning rate, we will increase (decrease, resp.) the learning rate multiplying it by the amount of learning rate change from the previous window, and vice versa. But, if the learning rate doesn not change for consectuive windows, and the performance degrades continuously or becomes zero, we record this. If the performance keeps degrading for a 10 consecutive window sizes [9], we reset the learning rate to the initial value. The objective behind is to make sure we restart the learning when the performance drops for a longer period. The learning rate is initialized randomly between 10^{-3} and 1.

The goal of the CACHEUS framework is to enable a single cache replacement policy that uses the combination of individual decisions taken by exactly two internal experts. Algorithm 1 depicts the generalized CACHEUS(A,B) algorithm with generic cache replacement experts, A and B. H_A and H_B are LRU lists of the history of items evicted by experts A and B, respectively, each of size $N/2$. Upon a cache hits, CACHEUS updates the internal data structures which includes moving the item to the MRU position of the cache and updating its frequency information. Upon a cache miss, CACHEUS checks the eviction histories for the requested item q, removes it from said histories, and updates the weights w_A and w_B. The weights are initialized to 0.5. Using the updated weights (Algorithm 2), CACHEUS chooses the expert (A or B) to use and obtains the eviction candidate accordingly, $A(C)$ or $B(C)$. Finally, CACHEUS updates its history, avoiding this update entirely if both experts suggest the same eviction candidate.

At the end of every window of N requests (N = cache size), CACHEUS updates its learning rate (Algorithm 3). First, the gradient of the performance (average hit-rate) with respect to the learning rate over the previous two windows is calculated. If the gradient is positive (negative, resp.), then the direction of change of the learning rate is sustained (reversed, resp.). The amount of gradient change determines the magnitude of the change in the learning rate. If the performance increases (decreases, resp.) by changing the learning rate, we will increase (decrease, resp.) the learning rate by an amount proportional to the learning rate change relative to the previous window. The learning rate is initialized randomly between 10^{-3} and 1. Finally, if the performance keeps degrading for a 10 consecutive window sizes [9], we reset the learning rate.

Like LeCaR, CACHEUS uses exactly two experts. The usage of more than two experts was considered for early CACHEUS versions. Interestingly, the performance with more than two experts was significantly worse than when using only LRU and LFU. Having multiple experts is generally not beneficial unless the selected experts are orthogonal in nature, and operate based on completely different and complementary strategies. The intuition here is that multiple experts will overlap in their eviction decisions thereby affecting learning outcomes and deteriorating the performance. We demonstrate in this paper that with two well-chosen experts CACHEUS is able to best the state-of-the-art with statistical significance.

4 Scan Resistance

Our initial experiments with CACHEUS using LRU and LFU as experts demonstrated inconsistent results when tested with a significantly wider range of workloads than the original LeCaR study did [34]. Of particular concern was the inability of CACHEUS(LRU, LFU) to handle the *scan* workload primitive type. Of the 5 different datasets comprising a total of over 329 different workloads that we examined, 4 of the datasets comprised *scan* workloads (see Table 2).

To understand the impact of *scan* on classic caching algorithms, we set up synthetic workloads that interleaved reuse with scan. Figure 5 shows performance versus cache size for two synthetic workloads wherein a single scan of size 60 items is interleaved between accesses to reused items. Let us assume that the *scan* phase is greater than twice the size of the cache (say 25). In this case, classic algorithms such as LRU evict resident items to absorb the new items anticipating their future reuse, giving up on hits for resident items that get reused beyond the scan phase.

State-of-the-art caching algorithms such as ARC, LIRS

Algorithm 1: CACHEUS(A, B)

Data: Cache C; Eviction histories H_A, H_B;
Weights w_A, w_B; Current time t;
Learning rate update interval i;
λ_t — learning rate at time t;
HR_t — average hit-rate at time t
Input: Requested page q
if $q \in C$ **then**
  C.UPDATEDATASTRUCTURES(q)
else
  UPDATEWEIGHT(q, λ, w_A, w_B)
  if $q \in H_A$ **then**
    H_A.DELETE(q)
  if $q \in H_B$ **then**
    H_B.DELETE(q)
  if C is full **then**
    if $A(C)$ == $B(C)$ **then**
      C.EVICT($A(C)$)
    else
      action = (A, B) w/prob (w_A, w_B)
      if *(action* == A*)* **then**
        if H_A is full **then**
          H_A.DELETE(LRU(H_A))
        H_A.ADDMRU($A(C)$)
        C.EVICT($A(C)$)
      if *(action* == B*)* **then**
        if H_B is full **then**
          H_B.DELETE(LRU(H_B))
        H_B.ADDMRU($B(C)$)
        C.EVICT($B(C)$)
  C.ADD(q)
if $(t\%i) = 0$ **then**
  UPDATELEARNINGRATE(λ_{t-i}, λ_{t-2i}, HR_t, HR_{t-i})

Algorithm 2: UPDATEWEIGHT(q, λ, w_A, w_B)

if $q \in H_A$ **then**
  $w_A := w_A * e^{-\lambda}$ // decrease w_A
else if $q \in H_B$ **then**
  $w_B := w_B * e^{-\lambda}$ // decrease w_B
$w_A := w_A/(w_A + w_B)$ // normalize
$w_B := 1 - w_A$

Algorithm 3: UPDATELEARNINGRATE(λ_{t-i}, λ_{t-2i}, HR_t, HR_{t-i})

$\delta_{HR_t} := HR_t - HR_{t-i}$
$\delta_{LR_t} := \lambda_{t-i} - \lambda_{t-2i}$
if $\delta_{LR_t} \neq 0$ **then**
  $sign := +1$ if $\frac{\delta_{HR_t}}{\delta_{LR_t}} > 0$, else -1
  $\lambda_t := \max(\lambda_{t-i} + sign \times |\lambda_{t-i} \times \delta_{LR_t}|, 10^{-3})$
  $unlearnCount := 0$
else
  if $HR_t = 0$ *or* $\delta_{HR_t} \leq 0$ **then**
    $unlearnCount := unlearnCount + 1$
  if $unlearnCount \geq 10$ **then**
    $unlearnCount := 0$
    $\lambda_t :=$ choose randomly between 10^{-3} & 1

and DLIRS each implement their own mechanisms for scan resistance. ARC limits the size of its $T1$ list used to identify and cache newly accessed items to preserve reused items in $T2$. Unfortunately, ARC's approach to scan-resistance makes it ineffective when handling the *churn* workload pattern. In particular, when a *scan* phase is followed by a *churn* phase, ARC continues to evict from $T1$ and behaves similar to LRU, as evidenced in one of our experiments (see Figures 10 and 11). Similarly, LIRS uses its stack Q to accommodate items that belong to the *scan* sequence. However, the size of Q is fixed to 1% of the cache , which cannot adapt to dynamic working sets. Finally, DLIRS reworks LIRS's solution by making Q adaptive. Despite its built-in adaptation mechanism, we note that DLIRS does not perform as well as LIRS in practice (see Figure 3).

4.1 SR-LRU

One policy that handles *scan* well is the classic Most Recently Used (MRU) policy. While LRU consistently evicts

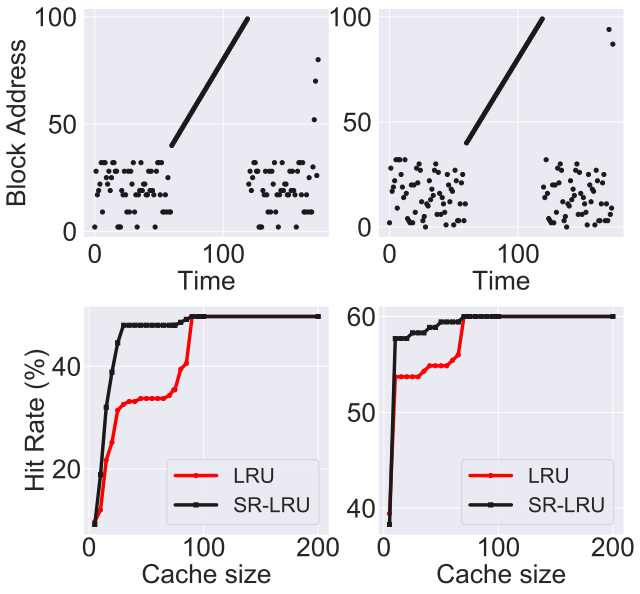

Figure 5: Motivating SR-LRU with the *scan* workload primitive type. *Two synthetic workloads are considered with 175 total requests and a single inserted scan: LFU-friendly pattern (left column) and LRU-Friendly pattern (right column). The size of the scan is 60 items in both cases.*

resident working-set items during scan, MRU evicts the previously inserted page placed at the top of the stack. We designed Scan-Resistant LRU (SR-LRU), an LRU variant that favors LRU friendly workloads while also being *scan* aware.

SR-LRU manages the cache in partitions similar to ARC

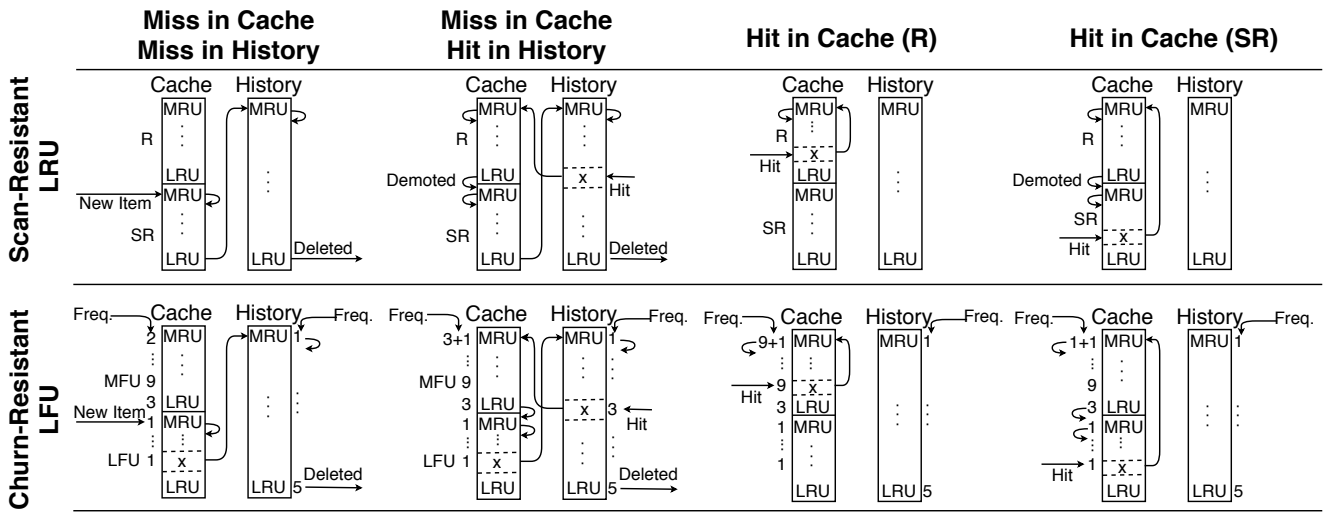

Figure 6: Understanding CR-LFU and SR-LRU. *Shown are actions taken to handle request x under:* cache miss*,* cache miss with *x* in history*,* cache hit with *x* in *SR, and* cache hit with *x* in *R.*

and LIRS. It divides the cache into two parts: one containing only items with multiple accesses (*R*) and the other for single access items as well as older items that have had multiple accesses (*SR*). The *SR* partition allows SR-LRU to be scan resistant; a partition for *new* items to be housed so that they do not affect the important items in *R*. SR-LRU only evicts from the *SR* partition — it evicts the LRU item of *SR* on a cache miss when the cache is full. Older items in *R* get *demoted* to *SR* to keep only important items that are being reused in *R*. In addition, SR-LRU maintains a history list *H* as large as the size of the cache that contains the items most recently evicted.

The basic workings of SR-LRU are as shown in Algorithm 4. We illustrate how a request for page *x* gets handled in Figure 6. On a cache miss where *x* is *not* in a history list, *x* is inserted to the MRU position of *SR*. Should the cache be full, the LRU item of *SR* is evicted to *H*, and should *H* be full the algorithm removes the LRU item of *H* to make space. On a cache miss where *x is* in *H*, *x* is moved to the MRU position of *R*. On a cache hit where *x* is in *SR*, *x* is moved to the MRU position of *R*. On a cache hit where *x* is in *R*, *x* is moved to the MRU position of *R*.

While SR-LRU could set a constant size for *SR* (similar to LIRS) and thereby be scan resistant, doing so would compromise its performance with LRU-friendly workloads for which *SR* is unfavorably sized [20]. Our approach to adapting SR-LRU is to adjust its partition sizes when we have found that SR-LRU either *demoted* or *evicted* incorrectly. If a demoted item gets referenced while in *SR*, SR-LRU infers that the size of *R* is too small and should be increased. To handle incorrect *evictions*, when an item is encountered for the first time, it gets marked as *new* after inserting it in cache. Should this item be evicted but then requested before it is removed from SR-LRU's history *H*, SR-LRU infers that the size of *SR* is too small to allow new items to be reused prior to being evicted. Items that enter the cache for the second time, after being placed in the history list previously, are not considered to be *new* items anymore.

Algorithm 4: SR-LRU

```
Data: Scan-resistant list SR; Reuse list R
C_demoted — count of demoted items in cache
H_new — count of new items in history
Input: requested page q
if q ∈ C then
    if q was demoted from R then
        δ = max(1, H_new / C_demoted)
        size_SR = max(1, size_SR − δ)
    R.MoveMRU(q)
else
    if q ∈ H then
        if q was new from SR then
            δ = max(1, C_demoted / H_new)
            size_SR = min(|C| − 1, size_SR + δ)
        H.Delete(q)
    if C is full then
        if H is full then
            H.Delete(LRU(H))
        H.MoveMRU(LRU(SR))
    SR.AddMRU(q)
UpdateSizes(SR, R)
```

To adapt itself, SR-LRU continuously computes a *target size* for *SR*. The algorithm reactively increases the size of *SR* upon hits in *H* by moving the LRU items of *R* into *SR* in order for *SR* to reach its *target* size. If the size of *SR* increases by too much, the demoted items being reused will inform the algorithm allowing it to reverse the erroneous increase.

The SR-LRU Difference Prior approaches to scan resistance are limited because they are either not adaptive (e.g.,

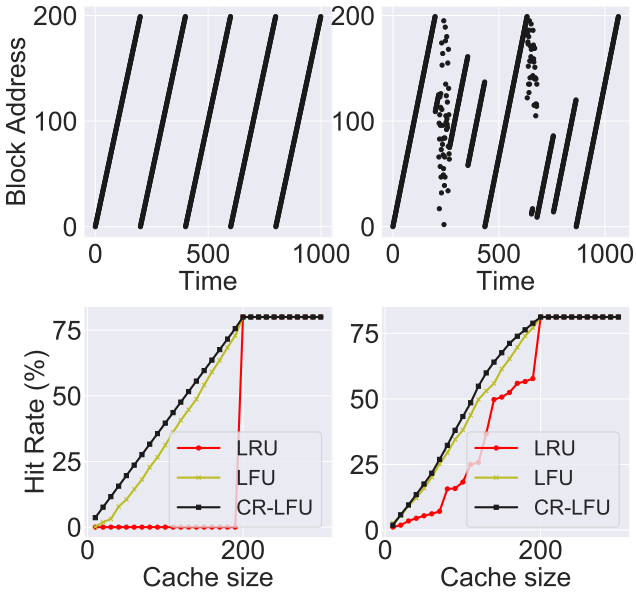

Figure 7: Motivating CR-LFU with the *churn* workload primitive type. *Two synthetic workloads are considered: a churn pattern (left column) and a combination of churn and LRU-friendly pattern (right column). The working set is 200 items.*

LIRS) or do not adapt well enough (e.g., DLIRS), or are unable to handle a scan followed by churn (e.g., ARC). The most important distinction in SR-LRU is balancing the need for being scan resistant with quickly recognizing when a workload is no longer *scanning*. In particular, SR-LRU tracks new items in history to distinguish between new items that belong to a scan from the new items that contribute to churn. As a result, SR-LRU continues to be effective immediately when a workload switches from scan to churn, as evidenced in our experiments (see Figures 10 and 11).

5 Churn Resistance

For the *churn* workload primitive type, if the number of items being accessed is larger than the size of the cache, an LRU-style algorithm would lead to churning of the cache content whereby items get repeatedly inserted into and evicted from the cache. On the other hand, the classic LFU assigns equal importance to all items with the same frequency. In a *churn* workload, all items have the same access frequency and these items may be accessed sequentially or otherwise. Other frequency-based algorithms like LRFU [18], that assign weights based on recency of access, result in LRU-based eviction for items with the same frequency; this unfortunately, does not prevent churning either.

Fortunately, a simple modification of the classic LFU turns out to be sufficient to handle the *churn* workload primitive type while continuing to retain the benefits of LFU. Churn-resistant LFU (CR-LFU) modifies the eviction mechanism in pure LFU by choosing the MRU (Most Recently Used) item to break the ties when several items have the least access frequency. By choosing the MRU item, CR-LFU effectively "locks" a subset of items with the lowest frequency into the cache, generating hits for the caching algorithm. Figure 6 illustrates the operation of algorithm CACHEUS using the SR-LRU and CR-LFU while handling a page request x in different situations.

We compare CR-LFU with LRU and LFU in Figure 7 for two different types of synthetic workloads: pure *churn* and mixed pattern of *churn* and *LRU-friendly*. Both LFU and CR-LFU outperform LRU when the cache size is less than the workload's working set size. Classic LFU evicts at random from among multiple items with the lowest frequency whereas CR-LFU evicts the MRU item. Because of that distinction, the average performance of CR-LFU is 8.67% and 3.83% higher than LFU for the *churn* and mixed pattern workloads respectively.

6 Evaluation

6.1 Experimental Setup

We conducted simulation-based evaluations of several state-of-the-art algorithms from the caching literature using publicly available production storage I/O workloads.

Algorithms: We compared CACHEUS against 6 previously proposed algorithms: *LRU, LFU, ARC, LIRS, LeCaR,* and *DLIRS*. In cases we could successfully contact the algorithm authors to obtain an implementation, we used the authors' original versions. In all other cases, we reimplemented the algorithms.

We also evaluated each of these against 3 variants of CACHEUS — **C1**: CACHEUS(ARC, LFU), **C2**: CACHEUS(LIRS, LFU) and **C3**: CACHEUS(SR-LRU,CR-LFU).

Workloads and Simulations. We used production storage I/O traces from 5 different productions systems for the simulation evaluation. Table 1 summarizes the workload datasets we used. A total of 17,766 simulations were conducted across 6 different cache sizes on 329 individual workloads contained within the 5 sets of workloads. Each individual workload represents an entire day of storage I/O activity from one storage system.

Cache Configurations. For evaluating caching algorithms, the primary metric of significance is cache hit-rate. To compare the relative performance of various caching algorithms, we chose caches that are sized relative to the size of each workload's *footprint*, i.e., all the unique data items accessed.

6.2 Time and Space Overheads

CACHEUS maintains roughly $2N$ pieces of metadata where N is the size of the cache, using N units to track cache-resident items and N additional units to track items that are in history. This is equivalent to state-of-the-art algorithms such as ARC and LIRS which each maintain approximately

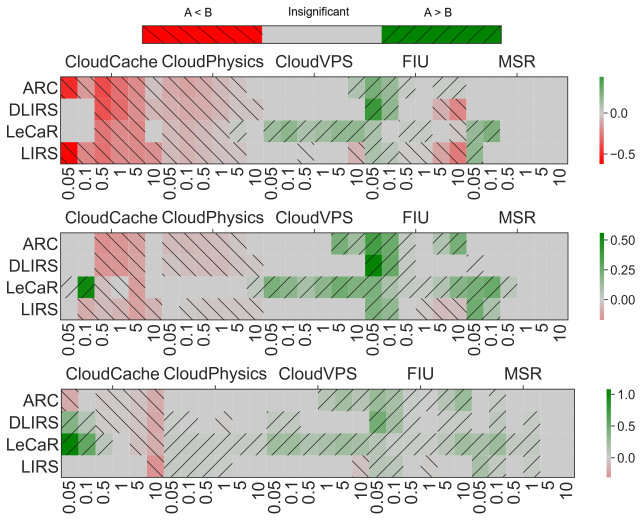

Figure 8: Paired t-test analysis to understand the difference in performance between (A) CACHEUS vs. (B) Other. *The three panels compare four "Other" algorithms (i.e., ARC, DLIRS, LeCaR, and LIRS) against the following variants of* CACHEUS*:* **Top:** *C1,* **Middle:** *C2* **Bottom:** *C3. Green colors indicate that the* CACHEUS *variant was significantly better, red colors indicate that the* CACHEUS *variant was significantly worse, and the gray color indicates no significant difference. Brighter green and red colors indicate higher effect sizes. Effect sizes were computed using Cohen's d-measure.*

N items of additional metadata to track a limited history. CACHEUS merges the additional metadata of individual experts (e.g. ARC, SR-LRU, and CR-LFU) and its own history for an effective size of N history items. Specifically, when SR-LRU and CR-LFU are used as experts in CACHEUS, the history metadata of each algorithm is reduced to $N/2$ for a total of N history metadata. The computational overhead of CACHEUS when it uses SR-LRU and CR-LFU as experts is bound by the computational overhead of LFU — $O(logN)$. This time complexity can be improved with a more careful implementation for LFU [30].

6.3 Statistical Analysis

We performed a broad palette of paired t-tests to evaluate the three CACHEUS variants against the strongest competitors across 17,766 experiments. A p-value threshold of 0.05 was used to judge statistical significance outcomes from the t-tests. Effect sizes were computed using the Cohen's d-measure, which measures the number of standard deviations that separate the two means. Figure 8 presents the results of our t-test analysis for the three CACHEUS variants.

To summarize the findings, C3 is distinctly the best performing algorithm in 47% of the workload-cache combinations with effect sizes ranging from 0.2 to 1.08 in 28% of the positive cases, is indistinguishable from the best performing state-of-the-art algorithm in about 40%, and is worse than the best performing algorithm for the remaining 13% with negative effect sizes of up to 0.31. For the 13% of the cases where an algorithm other than C3 is found to be distinctly better, no single algorithm is found to be consistently the best, indicating that C3 is an excellent choice overall. C2 is better than the best performing state-of-the-art in about 26% of the combinations with effect size in the range of 0.2 to 0.56 in 55% of the positive cases, indistinguishable from the best in 48% of the combinations, and worse in the remaining 27% of the cases with negative effect size of up to 0.17. C1 is better than the best performing state-of-the-art in about 20% of the combinations with effect size from 0.2 to 0.44 in 22% of the positive cases, indistinguishable from the best in 41% of the combinations, and worse in the remaining 39% of the cases with negative effect size up to 0.62.

We also analyze the best and worst case improvements in hit-rate for the best-performing CACHEUS algorithm, C3. Figure 9 presents the absolute difference in hit-rate for C3 relative to its competitors — ARC, LIRS, DLIRS and LeCaR, shown as a set of violin plots. Violin plots have the advantage of showing summary statistics, including the median, the quartiles and outliers along with a density shape for each Y-value [11]. The worst case degradation of 15.12% is observed with the MSR workload with cache size at 5% when compared against DLIRS. The best case improvement of 38.32% is observed with CloudPhysics workload at a cache size of 10% when compared against ARC.

6.4 Understanding CACHEUS

We focus our investigation and comparative analysis of CACHEUS against 3 of the best performing candidates: *(i)* **state-of-the-art adaptive** algorithm (ARC), *(ii)* **state-of-the-art scan-resistant** algorithm (LIRS); we do not consider DLIRS, its adaptive variant, which performs worse than LIRS on average, and *(iii)* **state-of-the-art machined-learned** algorithm (LeCaR), a predecessor of CACHEUS. To understand the performance advantage of CACHEUS, we measured hit-rates over time averaged across a sliding window equal to the size of the cache. In particular, we examine the performance for the webmail day 16 workload from the FIU trace collection. As shown in Figure 11, this workload includes a combination of multiple workload primitive types. For example, we observe a long *scan* for approximately 2 hours (between 6:30 and 8:30) followed by repeated accesses over a sub-set of the items (i.e., *churn*) for more than half the total workload duration.

6.4.1 CACHEUS C3 vs ARC

Figure 10 shows the performance over time for the four algorithms tested on webmail (day 16) workload. The total hit-rates for ARC, LIRS, LeCaR and C3 are 30.08%, 40.71% and 42.08% and 43.95% respectively. The leftmost plot shows the comparison against ARC. Initially a set of items that include a single large scan are accessed until the burst

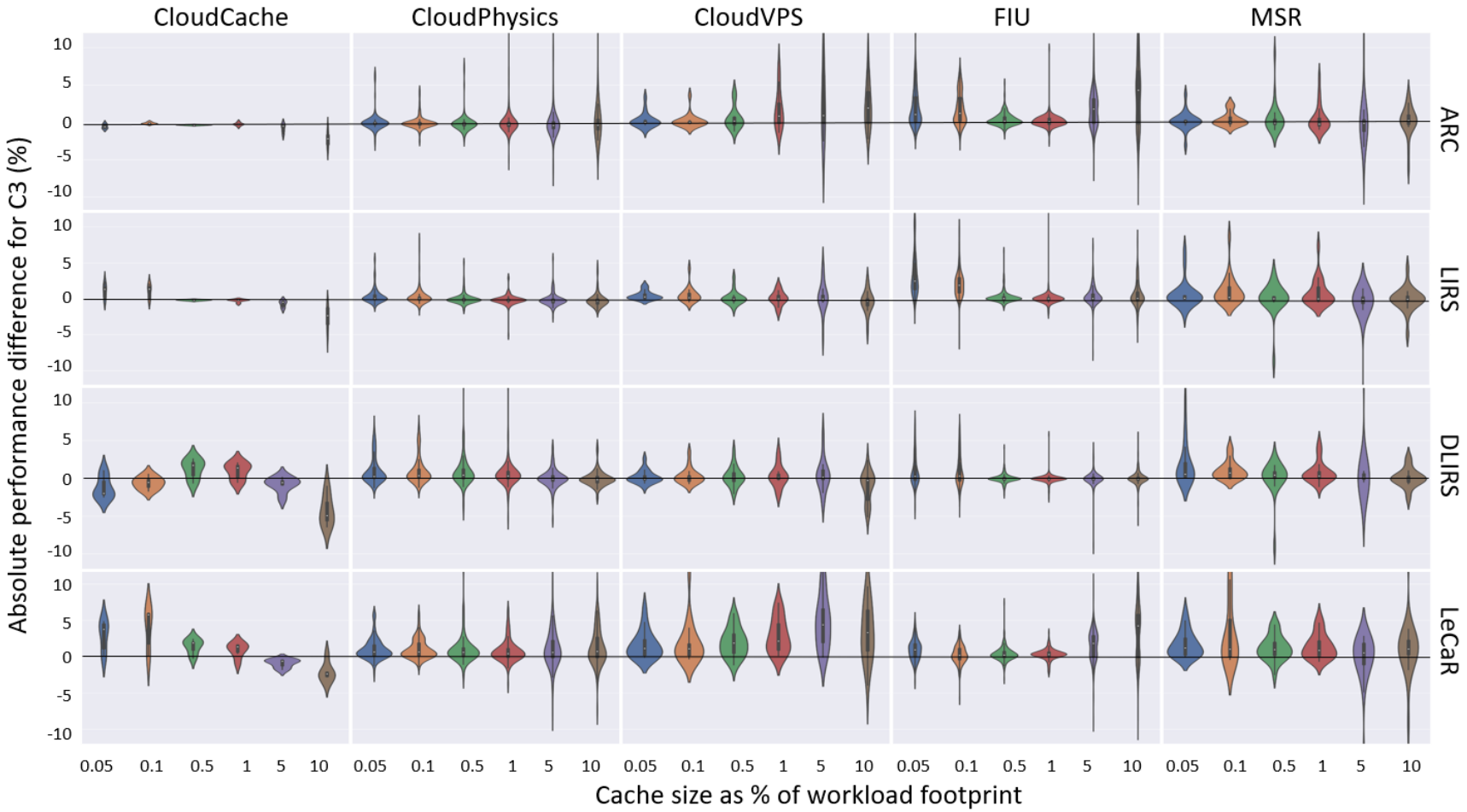

Figure 9: Absolute cache hit-rate difference distributions using CACHEUS algorithm C3 across workloads and cache sizes. *The figure displays four rows of violin plots with each row comparing the performance of CACHEUS C3 with the baselines of each row being the performance of ARC, LIRS, DLIRS, and LeCaR from top to bottom. Positive Y-values indicate that CACHEUS algorithm C3 performed better in comparison. The Y-range is truncated to the range (-12,12) for better readability, but with minimal loss of information. The violin plots show the median as a white dot, the range from the first to third quartile as a thick bar along the violin's center line, and a thin line showing an additional 1.5 times the interquartile range. It also shows the density shape at each Y-value [11], making these plots very informative.*

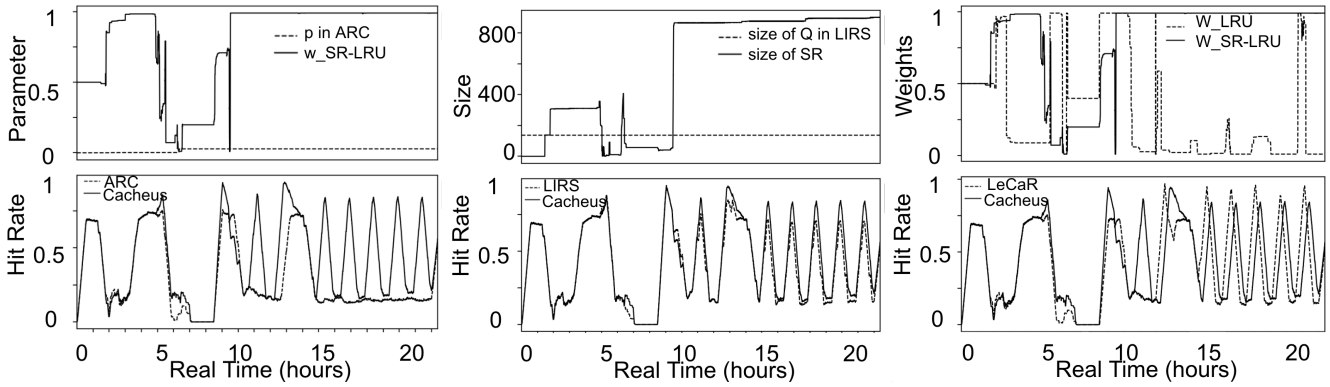

Figure 10: Detailed comparison of CACHEUS against ARC (left), LIRS (middle) and LeCaR (right) for the webmail (day 16) workload. *The lower plots show cache hit-rate computed using a sliding window equal to the size of the cache. The upper plot shows the internal parameter for each algorithm (p in ARC is normalized with respect to the size of the cache). Cache size is set to 10% of the workload footprint (54MB). The hit-rate improvements for* CACHEUS *with respect to ARC, LIRS, and LeCaR are 46.11%, 7.95% and 4.4% respectively.*

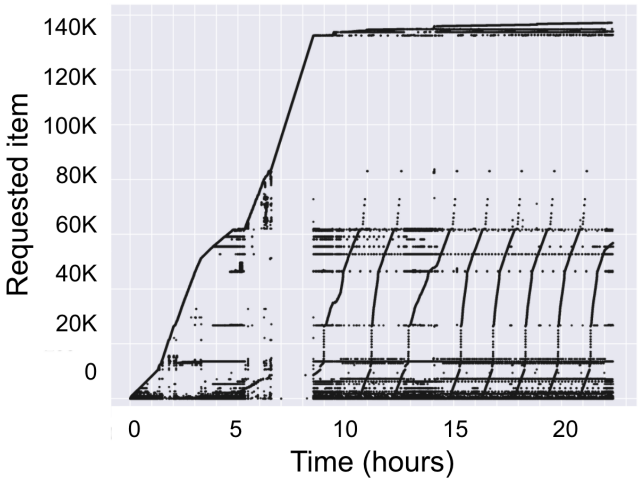

Figure 11: Access pattern for the webmail (day 16) workload from the FIU trace collection.

of unique accesses creates zero hits. C3 is able to maintain the previous working set in the cache, enabling it to generate hits post scan. ARC protects $T2$ as dictated by its internal parameter p close to 0 in an attempt to minimize cache pollution. Right after the scan finishes, a sequence of 8 churn phases starts to populate the cache. To respond effectively, ARC starts to increase the size of $T1$ to accommodate the new incoming items. However, the increments in p grow $T1$ slowly in steps of 1. In particular, during this entire trace ARC maintains its shadow list $B2$ empty by avoiding evictions from $T2$, even during churn. This behavior negatively impacts ARC's performance for the last 5 churn periods.

6.4.2 CACHEUS C3 vs LIRS

The center plot in Figure 10 compares LIRS and C3 using the same workload. LIRS uses a fixed size of Q equal to 1% of the cache size (138 items in this experiment). During the *scan* period, LIRS uses Q as a filter without affecting the working set previously populated in cache. For the *churn* phases, LIRS is able to keep in cache the important items by relying on its low-interference items in S. In particular, for *churn* phases, LIRS will always miss the first hits on the initial portion of the churn because these items will stay in cache for a short period of time. On the other side, C3 starts with a small size in SR to protect against the initial scan. During churn periods, C3 is able to dynamically accommodate new items in SR by increasing its size and therefore relaxing the scan protection. Finally, LIRS's ability to adapt to *LRU-friendly* workloads is limited by the size of Q.

6.4.3 CACHEUS C3 vs LeCaR

Finally, the rightmost plot in Figure 10 compares LeCaR and C3. The upper plot shows the weights for LRU and SR-LRU in LeCaR and C3 respectively, both initialized to 0.5. During the *scan* phase, LRU and SR-LRU get penalized due to the drop in performance until new hits in cache make them increase again. Even though choosing LFU is the right decision for LeCaR during *churn* phases, the delay in doing so prevents LeCaR of accumulating more hits than C3. In particular, C3 is able to capitalize during one *churn* period during the 11 hour while maintaining good performance for the last 7 hours of the workload. Most interestingly, towards the end when LeCaR mostly uses LFU, C3 exclusively relies on SR-LRU during *churn* periods. This is due to the fact that while SR-LRU was designed to handle *scan* phases, it also implements a way to avoid confusing churn periods with scan. This is done by marking items entering SR for the first time as *new* and keeping track of such items in H. If a *new* item is accessed again while in H, SR-LRU quickly corrects itself to disable scan protection.

7 Related Work

Past work on utilizing multiple experts within a cache replacement algorithm include ACME [1] and the follow up work on designing a master policy [10] which learned the weights of 12 distinct experts and used these to make eviction decisions. Since then, algorithms such as ARC [23], LIRS [13], DLIRS [20], and LeCaR [34] were developed and are considered the state-of-the-art.

CACHEUS builds on the successes of LeCaR. It improves upon LeCaR in a few ways. First, while LeCaR argued for using the classic LRU and LFU, CACHEUS demonstrates the importance of using more sophisticated experts. Second, CACHEUS simplifies LeCaR by identifying and eliminating redundant aspects of its machine-learning mechanism. Third, it creates a fully-adaptive version that is also lightweight. Finally, new lightweight experts, SR-LRU and CR-LFU improve upon LeCaR's experts to address two new workload primitive types, *scan* and *churn*. With these improvements, CACHEUS performs better than LeCaR as well as other state-of-the-art algorithms such as ARC, LIRS, and DLIRS.

SR-LRU is inspired by both ARC and LIRS. One important distinction between ARC and SR-LRU is that ARC evicts from either $T1$ or $T2$, while SR-LRU only evicts from a single spot: *SR*. Another distinction is SR-LRU's use of tags instead of separate histories ($B1$ and $B2$ in ARC) in order to enable reasonable adaptiveness. As to LIRS and its adaptive version, DLIRS, SR-LRU differs from these in the separation of history from internal partition/stack data structures, and its use of tags to determine relevance of items in history instead of explicitly pruning obsolete history items.

Recent works on adaptive caching include Least Hit Density (LHD) [4] which focuses on predicting an object's hits-per-space-consumed to determine evictions in a variable-sized object environment. LHD focuses on variable-sized caches of key-value stores or CDNs and was therefore not evaluated against the state of the art storage caches such as ARC and LIRS [4]. Like the state-of-the-art storage caching algorithms, CACHEUS is designed for a fixed-sized object caching environment and uses a novel reinforcement learning technique that engages exactly two complementary experts for significantly improving caching decisions.

8 Conclusions

Consistently high-performing caching continues to represent a fascinating, yet elusive, goal for storage researchers. CACHEUS serves this goal by creating a new class of lightweight and adaptive, machine-learned caching algorithms. The CACHEUS framework allows the use of exactly two, ideally complementary, experts to guide its actions. CACHEUS using the proposed new experts, SR-LRU and CR-LFU, is the most consistent algorithm for a range of workload-cache size combinations. Furthermore, CACHEUS enables easily combining a state-of-the-art caching algorithm such as ARC and LIRS with a complementary expert such as LFU to better handle a wider variety of workload primitive types. We believe that ML-based frameworks for utilizing caching experts holds great promise for improving the consistency and effectiveness of caching systems when handling production workloads. CACHEUS sources can be downloaded at `https://github.com/sylab/cacheus`.

Acknowledgments

We would like to thank the reviewers of this paper and our shepherd Ken Salem for insightful feedback that helped improve the content and presentation of this paper substantially. This work was supported in part by a NetApp Faculty Fellowship, and NSF grants CCF-1718335, CNS-1563883, and CNS-1956229.

References

[1] I. Ari, A. Amer, R. B. Gramacy, E. L. Miller, S. A. Brandt, and D. D. Long. ACME: Adaptive caching using multiple experts. In *WDAS*, pages 143–158, 2002.

[2] D. Arteaga and M. Zhao. Client-side flash caching for cloud systems. In *Proceedings of International Conference on Systems and Storage (SYSTOR)*, 2014.

[3] R. Battiti. Accelerated backpropagation learning: Two optimization methods. *Complex systems*, 3(4):331–342, 1989.

[4] N. Beckmann, H. Chen, and A. Cidon. LHD: Improving cache hit rate by maximizing hit density. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 389–403, 2018.

[5] J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305, 2012.

[6] J. S. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl. Algorithms for hyper-parameter optimization. In *Advances in neural information processing systems*, pages 2546–2554, 2011.

[7] L.-W. Chan and F. Fallside. An adaptive training algorithm for back propagation networks. *Computer speech & language*, 2(3-4):205–218, 1987.

[8] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12 (Jul):2121–2159, 2011.

[9] G. Einziger, O. Eytan, R. Friedman, and B. Manes. Adaptive software cache management. In *Proceedings of the International Middleware Conference*. ACM, 2018.

[10] R. B. Gramacy, M. K. Warmuth, S. A. Brandt, and I. Ari. Adaptive caching by refetching. In *Advances in Neural Information Processing Systems*, pages 1489–1496, 2003.

[11] J. L. Hintze and R. D. Nelson. Violin plots: A box plot-density trace synergism. *The American Statistician*, 52 (2):181–184, 1998.

[12] S. Huang, Q. Wei, D. Feng, J. Chen, and C. Chen. Improving flash-based disk cache with lazy adaptive replacement. *ACM Transactions on Storage*, 12(2):8:1–8:24, Feb. 2016.

[13] S. Jiang and X. Zhang. LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Proceedings of the ACM Sigmetrics Conference (SIGMETRICS)*, 2002.

[14] A. Khachaturyan, S. Semenovskaya, and B. Vainstein. Statistical-thermodynamic approach to determination of structure amplitude phases. *Sov. Phys. Crystallography*, 24(5):519–524, 1979.

[15] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *science*, 220(4598): 671–680, 1983.

[16] R. Koller and R. Rangaswami. I/O Deduplication: Utilizing content similarity to improve I/O performance. In *Proceedings of the USENIX Conference on File and Storage Technologies*, FAST, 2010.

[17] R. Koller, A. Verma, and R. Rangaswami. Generalized ERSS tree model: Revisiting working sets. In *Proceedings of IFIP Performance*, November 2010.

[18] D. Lee, J. Choi, J. H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE Transactions on Computers*, 50(12):1352–1361, Dec. 2001.

[19] A. Li, O. Spyra, S. Perel, V. Dalibard, M. Jaderberg, C. Gu, D. Budden, T. Harley, and P. Gupta. A generalized framework for population based training. *arXiv preprint arXiv:1902.01894*, 2019.

[20] C. Li. DLIRS: Improving low inter-reference recency set cache replacement policy with dynamics. In *Proceedings of the 11th ACM International Systems and Storage Conference (SYSTOR)*, 2018.

[21] N. Littlestone and M. K. Warmuth. The weighted majority algorithm. *Information and computation*, 108(2): 212–261, 1994.

[22] G. Loomes and R. Sugden. Regret theory: An alternative theory of rational choice under uncertainty. *The economic journal*, 92(368):805–824, 1982.

[23] N. Megiddo and D. S. Modha. ARC: A self-tuning, low overhead replacement cache. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2003.

[24] D. Narayanan, A. Donnelly, E. Thereska, S. Elnikety, and A. Rowstron. Everest: Scaling Down Peak Loads Through I/O Off-Loading. *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*, December 2008.

[25] V. Plagianakos, G. Magoulas, and M. Vrahatis. Learning rate adaptation in stochastic gradient descent. In *Advances in convex analysis and global optimization*, pages 433–444. Springer, 2001.

[26] H. Robbins and S. Monro. A stochastic approximation method. *The Annals of Mathematical Statistics*, pages 400–407, 1951.

[27] E. Rothberg, J. P. Singh, and A. Gupta. Working sets, cache sizes and node granularity issues for large-scale multiprocessors. In *Proceedings of the International Symposium of Computer Architecture (ISCA)*, 1993.

[28] S. J. Russell and P. Norvig. *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,, 2016.

[29] R. Santana, S. Lyons, R. Koller, R. Rangaswami, and J. Liu. To ARC or Not to ARC. In *Proceedings of the USENIX Workshop on Hot Topics in Storage Systems (HotStorage)*, 2015.

[30] K. Shah, A. Mitra, and D. Matani. An O(1) algorithm for implementing the LFU cache eviction scheme. `http://dhruvbird.com/lfu.pdf`, August 2010.

[31] W. L. Smith. Regenerative stochastic processes. *Proceedings of the Royal Society of London. Series A. Mathematical and Physical Sciences*, 232(1188):6–31, 1955.

[32] J. Snoek, H. Larochelle, and R. P. Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959, 2012.

[33] Storage Networking Industry Association. The SNIA's I/O Traces, Tools, and Analysis (IOTTA) Repository. `http://iotta.snia.org/`.

[34] G. Vietri, L. V. Rodriguez, W. A. Martinez, S. Lyons, J. Liu, R. Rangaswami, M. Zhao, and G. Narasimhan. Driving Cache Replacement with ML-based LeCaR. In

Proceedings of the USENIX Workshop on Hot Topics in Storage Systems (HotStorage), June 2018.

[35] C. A. Waldspurger, N. Park, A. Garthwaite, and I. Ahmad. Efficient MRC construction with SHARDS. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2015.

[36] Z. Wang, F. Hutter, M. Zoghi, D. Matheson, and N. de Feitas. Bayesian optimization in a billion dimensions via random embeddings. *Journal of Artificial Intelligence Research*, 55:361–387, 2016.

[37] C. C. Yu and B. D. Liu. A backpropagation algorithm with adaptive learning rate and momentum coefficient. *Proceedings of the International Joint Conference on Neural Networks*, 2:1218 – 1223, 2002.