# Agar: A Caching System for Erasure-Coded Data

Raluca Halalai, Pascal Felber, Anne-Marie Kermarrec, François Taïani

## To cite this version:

Raluca Halalai, Pascal Felber, Anne-Marie Kermarrec, François Taïani. Agar: A Caching System for Erasure-Coded Data. ICDCS 2017 - 37th IEEE International Conference on Distributed Computing Systems, Jun 2017, Atlanta, GA, United States. pp.1-11, 2017, <10.1109/ICDCS.2017.97>. <hal-01617146>

**HAL Id: hal-01617146**

**https://hal.inria.fr/hal-01617146**

Submitted on 16 Oct 2017

# Agar: A Caching System for Erasure-Coded Data

Raluca Halalai, Pascal Felber
University of Neuchâtel, Switzerland
Email: {first.last}@unine.ch

Anne-Marie Kermarrec
Inria, France
Email: anne-marie.kermarrec@inria.fr

François Taïani
U. of Rennes 1, IRISA / ESIR, France
Email: francois.taiani@irisa.fr

***Abstract*—Erasure coding is an established data protection mechanism. It provides high resiliency with low storage overhead, which makes it very attractive to storage systems developers. Unfortunately, when used in a distributed setting, erasure coding hampers a storage system's performance, because it requires clients to contact several, possibly remote sites to retrieve their data. This has hindered the adoption of erasure coding in practice, limiting its use to cold, archival data. Recent research showed that it is feasible to use erasure coding for hot data as well, thus opening new perspectives for improving erasure-coded storage systems.**

**In this paper, we address the problem of minimizing access latency in erasure-coded storage. We propose *Agar*—a novel caching system tailored for erasure-coded content. Agar optimizes the contents of the cache based on live information regarding data popularity and access latency to different data storage sites. Our system adapts a dynamic programming algorithm to optimize the choice of data blocks that are cached, using an approach akin to "Knapsack" algorithms. We compare Agar to the classical *Least Recently Used* and *Least Frequently Used* cache eviction policies, while varying the amount of data cached between a data chunk and a whole replica of the object. We show that Agar can achieve 16% to 41% lower latency than systems that use classical caching policies.**

## I. Introduction

Distributed storage systems represent the backbone of many cloud applications. They span geographically diverse regions in order to serve content to end users with *high availability* and *low latency*.

The first property, high availability, is achieved through redundancy. To that end, distributed storage systems either replicate data across multiple sites or use more elaborate approaches such as erasure coding [1]. *Replication* implies storing full copies of the data at different sites. This approach is simple to implement and provides good performance, as users can get data from the nearest data site. However, replication suffers from high storage and bandwidth overheads. By contrast, *erasure coding* achieves equivalent levels of data protection with less storage overhead, but is more complex to set up and operate. The key idea is to split an object into $k$ data blocks (or "chunks"), compute $m$ redundant chunks using an encoding scheme, and store these chunks at different physical locations. A client only needs to retrieve any $k$ of the $k + m$ chunks to reconstruct the object. Erasure coding is thus efficient in terms of storage space and bandwidth, but unfortunately incurs higher access latency, as users now need to contact more distant data sites to reconstruct the data.

The second property, low latency, is in turn attained in systems through *caching*. The principle consists in storing a subset of the available content in memory that is faster or closer to users than the original source. In the case of systems that span geographically diverse regions, caching decreases access latency by bringing data closer to end users.

The application of caching to erasure-coded data is, however, not obvious, as the presence of blocks offers many caching configurations that traditional caching policies such as LRU or LFU are unable to capture or exploit. More precisely, a caching mechanism for erasure-coded data should be able not only to decide *which* data items to cache, but also *how many blocks* to cache for each of them.

In this paper, we address this problem and consider how an intelligent caching mechanism can help minimize access latency in storage systems that span many geographical regions and use erasure coding to ensure data availability.

We propose Agar, a novel caching system developed specifically for erasure-coded data. The name Agar is a reference to an online game[1] in which each player controls a cell that aims to gain as much mass as possible at the expense of other cells. Similarly to the game, in our system, objects deemed valuable are able to "expand" in terms of the number of data chunks cached, causing less valuable objects to "shrink" or even disappear from the cache. Agar explores the trade-off between the storage cost of locally caching chunks of an object and the expected latency improvement. We adapt a dynamic programming approach designed for the "Knapsack" problem and use it to optimize the cache configuration.

In this work, we make the following new contributions:

- We propose an approach and an algorithm to optimize the cache configuration for erasure-coded data.
- We design and prototype a caching system based on `memcached` that monitors backend latency and uses our algorithm to reconfigure the cache. We make our code open source.[2]
- We integrate our prototype with Amazon Simple Storage Service (S3) and use the Yahoo Cloud Storage Benchmark (YCSB) to perform a thorough performance evaluation using read-only workloads.

The paper is organized as follows: in §II, we argue for the need for a caching strategy tailored to erasure-coded data. We then present the design of Agar, a caching system specifically developed for erasure-coded data, in §III. We describe our cache configuration algorithm in §IV. In §V, we compare our Agar prototype against alternative caching systems using the classic LRU and LFU cache replacement policies, while

[1] http://agar.io/

[2] https://github.com/ralucah/Agar-YCSB.git

varying the number of blocks kept in the cache. Finally, we discuss limitations and possible extensions in §VI, review related work in §VII, and conclude in §VIII.

## II. Background and Problem Statement

In this section, we discuss the rationale of designing a dynamic caching system dedicated to erasure-coded data. We first provide background on *the use of erasure coding in distributed storage systems* and *caching*, the go-to approach for improving performance. We then introduce a motivating example that illustrates a typical scenario where erasure-coded data is stored and cached at geographically diverse sites. Finally, we discuss how the problem of caching erasure-coded data maps to the Knapsack problem, and explain why a solution more complex than a greedy algorithm is needed.

### A. Erasure Coding in Storage Systems

Erasure coding splits data in $k$ blocks (or *chunks*), and computes $m$ redundant blocks using an erasure code. (We use a Reed-Solomon code in the rest of the paper.) This process allows clients of a storage system to reconstruct the original data with any $k$ of the resulting $k+m$ blocks. Erasure codes deliver high levels of redundancy (and hence reliability) without paying the full storage cost of plain replication. This conciseness has made them particularly attractive to implement fault-tolerant storage back-ends [1], [2].

Beyond the inherent costs of coding and decoding, however, erasure codes lead to higher latency, notably within geo-distributed systems. This is because they only partially replicate data, in contrast to a full replication strategy. Thus, they are more likely to force clients to access remote physical locations to obtain enough blocks to decode the original data. Because of this performance impact, some systems relegate erasure codes to the archiving of *cold*, rarely-accessed data and resort back to replication for *hot* data (e.g., Windows Azure Storage [3] uses this mechanism). Other systems, like HACFS [4], adopt different erasure codes for *hot* and *cold* data: a fast code with low recovery cost for hot data, and a compact code with low storage overhead for cold data. This dual-code strategy helps alleviate the computational complexity of decoding data, but HACFS still suffers from latency incurred by accessing data blocks from remote locations.

### B. Caching

The classical way to prioritize hot data over cold data is via a separate caching layer: caches store hot data and are optimized to serve that data quickly. Since cache memory is limited, caching works well for systems whose workloads exhibit a considerable degree of *locality*. Internet traffic typically follows a probability distribution that is highly skewed [5], [6], [7]. In particular, workloads from Facebook and Microsoft production clusters have shown that the top 5% of objects are seven times more popular than the bottom 75% [8]. This observation implies that a small number of objects are more likely to be accessed and would benefit more from caching. The *Zipfian* distribution has been widely used for representing such workloads; for instance, Breslau et al. [5] modeled the popularity of Web content using a Zipfian distribution.

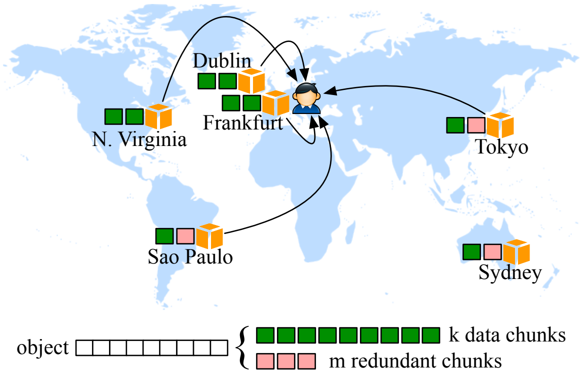


Fig. 1. An erasure-coded storage system spanning six AWS regions. Each region hosts an `S3 bucket` as persistent backend and a `memcached` server deployed on a large EC2 instance (2 vCPUs, 8GiB RAM). We populate the backend with 300 × 1 MB objects, encoded using a Reed-Solomon scheme with $k = 9$ data chunks and $m = 3$ redundant chunks (the total storage size, including redundancy, is thus 400 MB). The resulting twelve chunks are distributed among the regions in a round-robin manner, with each S3 bucket storing two data chunks.

### C. Motivating Example

To highlight the interplay between erasure coding and caching policies, we present a small experiment on a basic erasure-coded object store deployed on top of Amazon Web Services (AWS). Our goal is to show that one does not need to store complete copies of individual items to bring the full benefits of caching to erasure-coded data, but that selecting which blocks to cache is in fact non-trivial. This flexibility opens the path for advanced caching policies beyond traditional per-item strategies such as *Least Recently Used* (LRU) or *Least Frequently Used* (LFU), and motivates the approach we present in the next section. We show that:

- The improvement in latency is not a linear function of the number of cached data blocks; caching more blocks is not necessarily going to make the system faster.
- Finding the optimal number of blocks to cache for each object is difficult to achieve beforehand, as it depends on many external factors (requests distribution, network state, object popularity) that change over time.

Figure 1 shows the high-level architecture of our experiment. We use S3, Amazon's cloud storage service, to implement an erasure-coded object store spanning six AWS regions. To retrieve an object from the S3 backend, a client needs to contact at least five regions and retrieve at least $k = 9$ chunks.

Our experiment uses `memcached` [9] (a popular open-source in-memory cache engine that exposes a hash-table API) as a caching layer. `memcached` allows us to explicitly decide which chunks should be cached (and which not) in each individual AWS region. We allocate `memcached` enough memory in each region (500 MB) to accommodate our complete working set, in practice emulating an infinite cache.

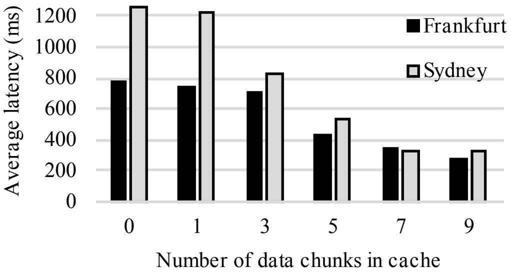


Fig. 2. Average read latency when caching a variable number of chunks. The relationship between the number of data chunks cached and the latency improvement obtained is non-linear.

This means that all requests for a given object are cache hits, except for the first one.

Using memcached's API, we vary the number of data chunks we retain in the cache of a region when an object is first retrieved by a client located in that region. Our goal in doing so is to better understand how the partial replication allowed by erasure coding impacts access latency.

Our experiment involves two clients: one in Frankfurt and one in Sydney. We use a customized version of the YCSB client [10] modified to support erasure-coding [11]. Each client performs 1,000 read operations on a pool of 300 × 1 MB objects. The read operations are generated from a Zipfian distribution with a skew exponent of 1.1. We measure the average latency experienced by both clients in 6 scenarios, in which we vary the number of data chunks $c$ retained in local cache instances for each object, from $c = 0$ to $c = k = 9$.

The first scenario ($c = 0$) is an extreme baseline case that does not use the caching layer: clients directly read data chunks from the backend and decode them. The remaining scenarios store $c$ chunks of each retrieved object in the memcached instance of the region the request originates from, for $c \in \{1, 3, 5, 7, 9\}$. The most distant chunks are cached first for each object, in effect progressively decreasing in each experiment the number of Amazon regions clients must access to reconstruct an object.

The results of our experiment (Figure 2) show that gains in latency are not proportional to the number of chunks retained in the cache.

- If only a few chunks are cached (e.g., up to 3 for Frankfurt), benefits remain minimal. This is because in this case, the overall latency is dominated by the last and slowest chunk that the client still needs to retrieve from a remote location.
- Conversely, once a critical mass of cached chunks has been reached (e.g. 7 chunks for both clients), caching more chunks brings only minimal returns. This is because the latency incurred by the slowest block(s) is masked by the delay required to retrieve the closest block(s).

Where these turning points lay further depends on the position of the client, and the latency and bandwidth experienced between regions: Sydney can already greatly benefit from 3 locally cached chunks, while this level of caching makes very little difference to Frankfurt.

The above observations carry important lessons for caching policies applied to erasure-coded storage systems: the values of chunks are not equal; for instance, it might be better to store 3 chunks for most objects in Sydney (thus *partially* caching the corresponding objects), than 9 chunks for a few objects. Unfortunately, any caching policy that reasons at the level of objects is unable to make such choices, which calls instead for *block-aware approaches* optimized for erasure-coded data.

### D. A Cache Dedicated to Erasure-Coded Data

The problem of caching erasure-coded data can be interpreted as a variant of the century-old Knapsack problem [12], which seeks to maximize the value of a set of elements packed into a container of bounded capacity. In our scenario, the container is a local cache, elements are blocks, and the value of individual blocks corresponds to the overall latency improvement that local clients will perceive over the entirety of their requests if this block is cached, i.e., how much faster it will be for clients to retrieve the data item from the cache instead of the backend. The *weight* of a block is the space it occupies in the cache.

In the absence of erasure coding, choosing which data items to cache in a storage system is an instance of the 0/1 Knapsack problem, where each element is unique (i.e., you cannot put more than one of each element in the knapsack). Greedy algorithms do not generally work well for 0/1 Knapsack and can err by as much as 50% from the optimal value [13].

However, by splitting data items into chunks, erasure coding greatly increases the complexity of the corresponding Knapsack problem. The problem might at first appear related to Fractional Knapsack [14], where it is possible to put a fraction of an element into the knapsack (and for which greedy algorithms yield optimal solutions). However, (i) *the non-linear dependency between fractions of value* (latency improvement) *and weight* (cached blocks) (as shown in §II-C), and (ii) *the finite choices of fractions* (i.e., options dictated by the choice of erasure coding parameters) make the problem closer to 0/1 Knapsack than to Fractional Knapsack, and introduce the need for a tailored algorithm to select which blocks to cache. In the following, we therefore propose to adapt a dynamic algorithm solution used for 0/1 Knapsack to the problem of caching erasure-coded blocks.

## III. Design

This section introduces Agar, a caching system we specifically developed for erasure-coded data. We designed Agar around the use-case described in §II-C: an erasure-coded object store deployed across several regions. Agar maintains independent caches in each region, along with components that implement a dynamic caching algorithm.

Unlike a caching eviction policy that decides which object to remove from the cache, Agar estimates the *popularity of individual objects*, as well as *potential latency gains* in order to *pre-compute* a static cache configuration that will be used during a fixed period; this period is a system parameter, and depends on how rapidly access patterns are expected to change. Agar's design exploits three core assumptions:

- Access patterns vary across regions, so caches from different regions require different configurations, and do not require coordination.
- Access patterns vary over time, so we need to periodically recompute the configuration of each individual cache.
- Individual objects do not have to be read entirely from the backend or entirely from the cache. Thus, Agar supports partial caching, and can benefit from partial cache hits.

The goal of Agar is to find a good trade-off between the *number of chunks to cache* and *the overall latency improvement* for each object. Latency improvement depends on the position of the client relative to the servers that store the content of interest and the access trend in the nearest region. Similarly to the LFU cache eviction policy, Agar requires statistics regarding object popularity and an estimation of the latency cost incurred when reading data chunks from individual regions.

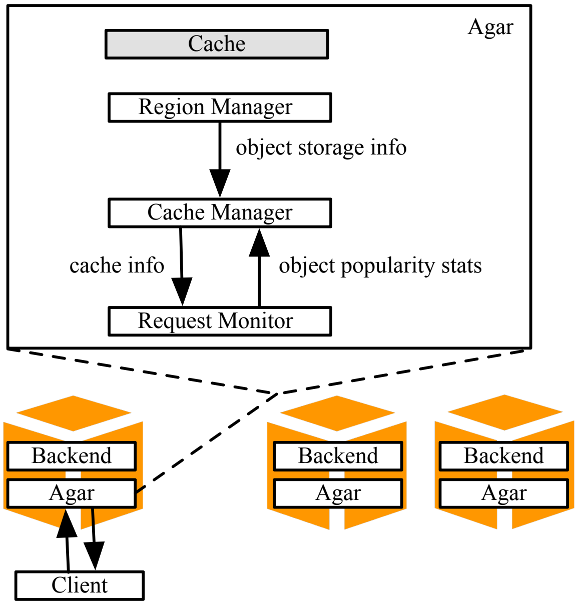


Fig. 3. Design of Agar. We show how Agar integrates with a typical erasure-coded storage system and zoom in on the components of an Agar region-level deployment.

Figure 3 provides an overview of the components of Agar and shows how they fit within a typical erasure-coded storage system. We envision that Agar has nodes deployed within most of the regions that the storage system spans (represented as orange cubes in Figure 3). In our current design, Agar nodes from different regions do not collaborate with each other. On these nodes, Agar sits as a layer between clients and the region's backend instance, and contains four key components (zoomed-in frame in Figure 3):

*a) Region Manager:* maintains a high-level overview of the storage system's topology, i.e., the regions that it spans and the policy it uses to distribute data chunks among regions. (In this paper, we assume a round-robin distribution policy.) The region manager periodically measures how much it takes to read a data chunk from each region and uses this information to estimate the latency improvement that clients would get if blocks from that region were cached locally, thus removing the need to access that region.

*b) Request Monitor:* listens for client requests and computes statistics regarding the popularity of each object over a predefined time interval. Agar uses an exponentially weighted moving average to keep track of popularity over time (more details in §IV). Before a client reads an object, it contacts the request monitor asking for hints regarding where to get the data chunks necessary to reconstruct that object. The request monitor forwards such requests to the cache manager and updates the statistics on object popularity.

In our current design, the request monitor is involved in each operation that clients perform. This did not increase latency in our experiments because Agar resides geographically close to the clients. For large deployments, we believe that techniques like TinyLFU's [15] approximate access statistics can avoid the request monitor becoming a bottleneck, while maintaining similar effectiveness.

*c) Cache Manager:* periodically computes the ideal cache configuration (i.e., what objects should be cached and how many chunks for each) based on object popularity statistics from the requests monitor and information about the system's backend deployment from the region manager. It provides hints regarding what data chunks should reside in the cache to the requests monitor, which forwards them to the client. The cache manager runs our dynamic algorithm to compute the ideal cache configuration (described in §IV).

*d) Cache:* provides volatile bounded storage where clients store chunks of erasure-coded data, according to the hints received from the requests monitor.

## IV. Algorithm

This section describes the algorithm that Agar's cache manager runs periodically in order to propose a static cache configuration. In more details, the cache manager chooses *which data objects to cache* and *how many chunks to store for each of them.*

As explained in §II-D, we map the problem of choosing which data chunks to cache to a Knapsack optimization problem. We compute the *weight* of a cached item as the amount of space that it would occupy in the cache (i.e., the number of its chunks that are cached), while the *value* is the overall latency improvement that caching the respective blocks would bring to the system.

Our caching algorithm works in two steps: 1) it first *generates caching options*, and then 2) it chooses a subset of caching options to *define the cache contents.*

### A. Generating Caching Options

A *caching option* is a hypothetical configuration that captures the implications of caching a specific set of chunks for an object. Each caching option contains:

- a key identifying the object it corresponds to;
- a set of data chunks to cache for that object;
- a weight, given by the number of data chunks to cache;
- a value, computed as the overall latency improvement that caching this set of blocks would bring to the system.

For each object known to the request monitor, we iteratively generate caching options. (For the sake of simplicity, we have not included the pseudo-code for this step.) Each caching option includes a weight, which varies between 1 and $k$ data chunks. (Recall that a client needs exactly $k$ chunks to reconstruct an object, so it does not make sense to cache more.) The algorithm needs to choose data chunks to put in each of the configurations. The cache stores the blocks that would be retrieved in the common case by the client, i.e., a client would not attempt to retrieve the furthest $m$ blocks unless there are failures. Thus, the algorithm first discards the $m$ blocks that are furthest away from the cache in terms of latency, because in the common case (without failures) those would not need to be accessed by clients. Caching items implies downloading them a priori; therefore, we optimize the latency penalty incurred by a cache miss by not caching the furthest blocks. The configurations are then filled with data blocks, from the most distant remaining data sites, until reaching the associated weight.

```
function POPULATE($Keys$, $AllOptions$, $CacheSize$)
    ▷ AllOptions — set of caching options for all keys
    ▷ Keys — set of keys sorted in decreasing value order
    ▷ CacheSize — available cache size
    ▷ MaxV — associative array [$Size$] $\rightarrow$ $Config$

    $MaxV[0] \leftarrow$ EMPTYCONFIG()                ▷ Initial state

    ▷ Iterate through keys in decreasing value order
    for $Option \in$ ORDERBY($AllOptions$, $Keys$) do
        ▷ Improve config but keep the same weight
        for $Config \in MaxV$ do
            RELAX($Config$, $Option$, $AllOptions$)
        end for

        ▷ Improve config by adding option at the end
        for $Config \in MaxV$ do
            Let $W = Config.Weight + Option.Weight$
            Let $V = Config.Value + Option.Value$
            Let $C = MaxV[W]$        ▷ Add new if missing
            if $C.Value < V$ then
                ADDTOCONFIG($C$, $Option$)
            end if
        end for
    end for

    return $MaxV[CacheSize]$
end function
```

Fig. 4. Algorithm that computes a static configuration for the cache, based on the weight and value of each caching option.

After choosing the set of blocks in each caching option, the algorithm needs to compute the associated values. To that end, we compute an estimation of the overall latency improvement that adopting a certain caching option would bring. This is computed as $popularity \times latency\_improvement$.

We compute the popularity of individual objects using an exponentially weighted moving average:

$$popularity^{i}_{key} = \alpha \cdot freq^{i}_{key} + (1 - \alpha) \cdot popularity^{i-1}_{key},$$

where $key$ identifies the object to which the caching option corresponds, $i$ indicates the time period when this computation is done, $freq^{i}_{key}$ is the access frequency for that object in the current time period, and $\alpha$ is the weighting coefficient (0.8 in our experiments).

To compute the *latency improvement*, the algorithm needs to know the latency to each backend region. The region manager computes this by retrieving several data blocks from each region in a warm-up phase. Using this data, the algorithm computes the *latency improvement* as the difference between the latencies to the most distant region that is contacted when the set of blocks in a caching option are cached versus when they are not. This assumes that the client requests blocks in parallel and that the requests do not interfere with each other.

TABLE I
READ LATENCY FROM THE POINT OF VIEW OF FRANKFURT.

| Frankfurt | Dublin | N. Virginia | Sao Paulo | Tokyo | Sydney |
|---|---|---|---|---|---|
| 80 ms | 200 ms | 600 ms | 1,400 ms | 3,400 ms | 4,600 ms |

*Example:* We consider the deployment in Figure 1 and assume our algorithm is running on an Agar node hosted in Frankfurt. The algorithm first estimates the latencies of getting blocks from each backend region, shown in Table I. There are five different caching options possible for an object identified by $key_1$, storing 1, 3, 5, 7, and 9 blocks, respectively. For the option with weight 1, the algorithm chooses to cache the block from Tokyo. Since the client only needs $k$ blocks to reconstruct the object, our algorithm discards the $m = 3$ blocks that are furthest away: two from Sydney and one from Tokyo. To compute the value corresponding to this caching option, it first takes the popularity of $key_1$. Suppose for simplicity that this is the first iteration of the algorithm, so the previous popularity is 0, and the current frequency of $key_1$ is 100. Thus, the popularity is 80 $(0.8 \times 100 + 0.2 \times 0)$. The estimated latency improvement is 2,000 ms, computed as the latency difference between the furthest region contacted when the block is not cached (Tokyo) and the furthest one contacted when the block is cached (Sao Paulo). Then, the value is $80 \times 2{,}000 = 160{,}000$. Similarly, for option 2 the algorithm chooses to cache blocks from Sao Paulo and Tokyo, and the value is $80 \times (1{,}400 - 600) = 64{,}000$.

### *B. Choosing the Cache Contents*

Our algorithm uses the generated caching options to compute a cache configuration. Figure 4 shows the pseudocode for choosing the contents of the cache. The algorithm uses

```
function RELAX($Config$, $Option$, $AllOptions$)
    $BestConfig \leftarrow Config$
    for $OldOption \in Config.Options$ do
        ▷ Replace $OldOption$ with alternative option $O$ for the same key, but with a lower weight $W$, making room for $Option$
        Let $W = OldOption.Weight - Option.Weight$
        Let $O =$ SEARCHOPTION($AllOptions$, $W$, $OldOption.Key$)
        Let $V = Config.Value - OldOption.Value + O.Value + Option.Value$
        if $BestConfig.Value < V$ then
            $BestConfig \leftarrow$ REPLACEANDADD($Config$, $OldOption$, $Option$)
        end if
    end for
    $Config \leftarrow BestConfig$
end function
```

Fig. 5. Relaxation function that improves a configuration's value without increasing its total weight.

a dynamic programming approach: it computes intermediate configurations for subsets of objects and then iteratively improves these intermediate solutions as it considers new objects.

In Figure 4, $MaxV$ is an associative array storing intermediate cache configurations. In particular, $MaxV[size]$ holds the best configuration discovered so far for a cache of size $size$. New configurations are implicitly created upon first access to a key, i.e., if no configuration has yet been stored under $MaxV[size]$, an empty configuration is added on-the-fly on line 17. There are two methods through which intermediate configurations are improved:

1) *Relaxation* (Figure 4, lines 10–12). The concept of relaxation is similar to the one in graph theory (e.g., Dijkstra's algorithm). RELAX checks if a new caching option *Option* can replace an existing one in an intermediate configuration, yielding a better overall value. The replacement can be *total*: an object already in the configuration is completely evicted in favor of the new option; or *partial*: the old option is only partially evicted, having fewer blocks in the configuration. Figure 5 shows the pseudocode for the relaxation method.
2) *Addition* (Figure 4, lines 14–21). The ADDTOCONFIG method (Figure 4, line 19) adds a new option at the end of an existing configuration, increasing its weight. If there is already another configuration with this new weight, it is replaced only if it has a lower value.

## V. EVALUATION

We built a prototype of Agar and used it to perform a thorough quantitative evaluation. We compare Agar against caching systems that use the classical Least Recently Used (LRU) and Least Frequently Used (LFU) cache replacement policies, while varying the amount of data kept in the cache from one chunk to a whole replica (i.e., $k$ chunks).

We first describe our experimental setup in §V-A. Then we answer the following questions:

- How does Agar compare to other caching policies (§V-B)?
- How do the cache size and workload influence the performance of Agar (§V-C)?
- What do the cache contents look like in Agar (§V-D)?

### A. Experimental Setup

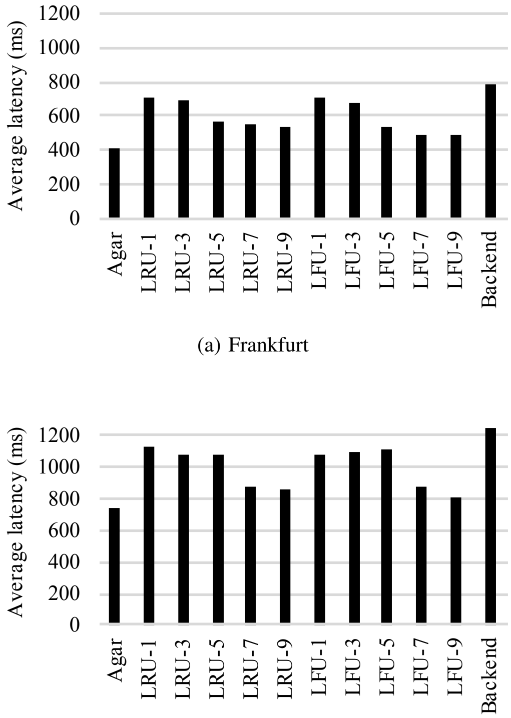
We implemented our Agar prototype in Java, and integrated it in the deployment shown in Figure 1. For our experiments, we modified the YCSB client [10]:

- First, we added support for erasure coding via the Longhair library [11]. On a write operation, the client encodes the object and writes the resulted chunks to S3 buckets concurrently. On a read operation, the client requests data chunks in parallel and, after it has received $k$ chunks, it decodes them. The read latency measured by our modified YCSB client accounts for reading a full object, and not just a chunk.
- Second, we added support for Agar. The YCSB client communicates with Agar in order to know what regions to contact. The client is also responsible for writing data to caches. This operation does not impact the latency measurements, as it is done in a separate thread pool, and not concurrently with reads. We deploy YCSB clients on large EC2 instances in the same regions as Agar.

We use several customized versions of the YCSB client, which differ in terms of reading strategy:

- *Agar*—reads content via our Agar caching system.
- *Backend*—reads content directly from the S3 buckets.
- *LRU*—reads content via a cache that stores a predefined number of erasure-coded chunks for each data record and supports the *Least Recently Used* policy. For our experiments, we rely on `memcached`'s LRU policy.
- *LFU*—reads content via a cache that stores a predefined number of erasure-coded chunks and supports the *Least Frequently Used* cache replacement policy. This client includes an additional proxy component that tracks request frequency for each object.

Unless stated otherwise, we use a read-only workload that follows a Zipfian distribution with skew factor 1.1, a cache



(a) Frankfurt

(b) Sydney

Fig. 6. Average read latency when using Agar vs. LRU- and LFU-based caching systems vs. Backend.

size of 10 MB—which fits ten full objects (9 chunks each), and set the cache reconfiguration period to 30 seconds for Agar and LFU. All results represent averages of 5 runs. Each run contains 1,000 reads. Each YCSB instance is configured to run 2 clients; each client uses a thread pool to make requests for chunks in parallel.

### B. Agar Compared to Other Caching Policies

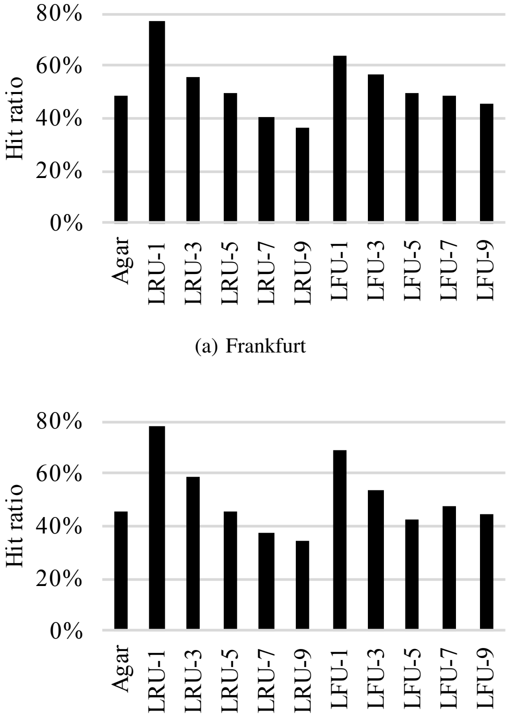
In this experiment, we compare the latency and cache hit ratio of Agar to those of the LFU and LRU policies with fixed number of chunks. We show that Agar can use the cache more efficiently, obtaining better performance than classical policies.

As shown in §II-C, the trade-offs regarding the number of chunks to store are different, depending on the region where the client runs. Therefore, we run this experiment using clients in two regions: 1) In the first scenario, we deploy our clients at Frankfurt, which has a relatively central position in our deployment, and is rather close to another region: Dublin. 2) In the second scenario, we deploy our clients at Sydney, which represents the opposite of Frankfurt, being far away from all other regions.

Figure 6 shows how Agar compares to LRU, LFU, and the backend, in terms of average read latency. Agar adapts to the particularities of each site and consistently outperforms the classical policies.

In Frankfurt, Agar obtains 15% lower latency than LFU-7, which is the next best policy (Least Frequently Used, caching 7 chunks for each object). Agar has an average latency of 416 ms versus LFU-7's 489 ms. When compared to the worst-performing setup, LRU-1, Agar yields 41% lower latency.

In Sydney, Agar obtains 8.5% lower latency than LFU-9, which is the next best policy. Agar obtains a latency of 736 ms, versus LFU-9's 803 ms.



(a) Frankfurt

(b) Sydney

Fig. 7. Hit ratio when using Agar vs. LRU- and LFU-based caching systems vs. Backend.

We also examined the hit rates that the different policies obtained in this experiment. Figure 7 shows the hit ratio, computed as the number of cache hits – *total hits* (all blocks were read from the cache) or *partial hits* (only a subset of blocks were read from the cache) – divided by the number of requests issued.

As expected, storing fewer chunks per object leads to higher hit rates, as high as 76%. However, storing fewer chunks leads to low overall latency improvement. Agar finds a good trade-off between storage cost and latency improvement for each object, storing more chunks for popular items, but then reducing the number of chunks corresponding to less popular items. Overall, Agar's hit ratio is higher than the hit ratio of the LRU and LFU policies storing 7 or 9 chunks for each object (§V-D has some more insight on how Agar manages its cache).

In this experiment, we showed that Agar outperforms classical policies like LRU or LFU. Moreover, unlike static policies, Agar can adapt to the particularities of the workload. Agar obtains this performance gain over LRU and LFU by carefully managing the trade-off between the size each object occupies in the cache and the overall latency improvement.

### C. Influence of Cache Size and Workload

In this experiment, we study the impact of external factors on the performance of Agar and its competitors—LRU and LFU. In the previous experiment, we kept the cache size and workload pattern fixed, while in this experiment we vary them to evaluate how the different policies react. We run this experiment using clients deployed at Frankfurt.

We vary the cache size between 5 MB (fits 5 full objects) and 100 MB (fits 100 full objects), while keeping the workload fixed (Zipfian, with skew 1.1). Figure 8a shows the average read latency.

When the cache is very small, there is little room for optimization, but Agar can still outperform all alternatives by more than 6.5%. As cache size increases, so does the advantage of Agar: it obtains 15% lower latency than alternatives for 10 MB cache size, and 16% for 20 MB. When the size increases beyond that, the cache becomes large enough to fit all popular data and Agar's lead starts to decrease: 12% for 50 MB and 1% for 100 MB. Overall, Agar outperformed LFU and LRU over a wide range of deployment scenarios, with the cache size ranging from 1% to 25% of the backend (5 to 100 MB).

Next, we keep the cache size fixed at 10 MB and vary the workload. First, we experiment with a workload that follows a uniform request distribution. Next, we experiment with Zipfian workloads with different skews; the skew is the coefficient that determines the number of popular data elements: higher skew means that fewer and fewer items become increasingly popular. We show in Figure 9 how the skew influences the popularity of the objects in the workload. Since the object size is 1 MB, it is also easy to see how much of the total data (300 MB) can fit in a cache of a given size (the horizontal axis can be interpreted as cache size as well).

Figure 8b shows the average read latency of Agar and its alternatives for different workloads. When the workload follows a uniform distribution, all clients perform similarly. The cache hit rates are very small because all data items are equally popular; therefore, the choice of caching policy makes no significant difference. When the skew of the Zipfian distribution is low, the workload pattern is similar to a uniform distribution and thus, the same effect is observed.

As the skew of the Zipfian distribution increases, however, some elements become more popular and caching them has a higher impact on the overall latency. Agar and LFU are the quickest to benefit from this type of workloads and can lead to lower overall latencies, with Agar taking a 5.8% lead for skew 0.8, 7.2% for 0.9, 13% for 1.0, and peaking at 15% for 1.1. As the skew becomes larger, only a small subset of objects account for most of the reads in the workload, and LFU-9 can catch up to Agar, since all of the highly popular items can fit in the 10 MB cache. At skew 1.4, the lead of Agar starts to decrease, dropping to 14%.

In this experiment, we showed how Agar compares to LRU and LFU when the workload distribution and the cache size vary. Agar consistently outperforms static policies when *system designers are cost-conscious* and *cache size is limited.*

### D. Cache Contents

In this experiment, we take an inside look into how Agar manages the cache contents. We take snapshots of the data that Agar chooses to cache for clients running in Frankfurt and Sydney, and for cache sizes of 5 MB and 10 MB.

Figure 10 shows the distribution of object sizes in Agar's cache. There are several interesting aspects to note. First, Agar diversifies the contents of the cache, rather than having the majority of the cache filled by a certain object size. Second, for each scenario Agar chooses to manage its cache differently. This again argues for a dynamic policy, such as Agar. Finally, despite the diminishing returns of storing entire replicas (9 blocks in cache), Agar chooses to allocate a significant fraction to this. This is explained by the high skew, which means that a few objects are so popular that the difference between disk and memory latency becomes important.

## VI. Discussion

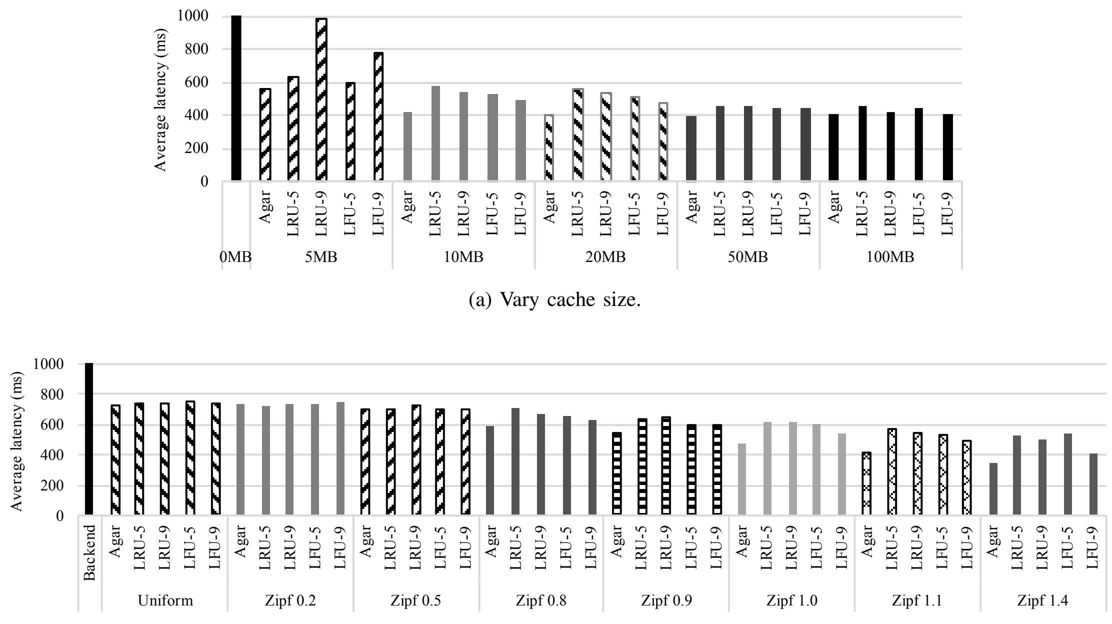
In this work, we focus on how Agar's caching algorithm can improve latency. In our evaluation (§V), we show that this algorithm can bring significant improvements—16% improvement over the next best policy we compare to. However, going from a research prototype to a full system requires significant engineering. In this section, we quickly address some of the unanswered questions that still need to be investigated.

While improving latency is important, Agar needs to also address throughput in order to scale. We did not yet thoroughly investigate the question of throughput and the potential for the cache manager and requests monitor to become bottlenecks in the system, negating the benefits of the better cache configuration. We believe that there are no fundamental hurdles in Agar that would prevent its scalability.

The request monitor in Agar is similar to the statistics components of other LFU caching systems and the same engineering optimizations apply. For example, in our prototype, we use UDP messages for communication between the clients and the request monitor, to minimize overhead. In our evaluation, we measured the average time for Agar's request monitor and cache manager to process a client request to be 0.5 ms.

The cache manager needs to periodically run the algorithm to choose a cache configuration. In our prototype, that time is $O(C^2)$, where $C$ is the cache size. We noticed that the configuration for a given cache size stabilizes soon after the first configuration is obtained by the dynamic programming algorithm (soon after $MaxV[C]$ is first obtained). Based on this, we optimized the implementation to stop execution a fixed number of iterations after first obtaining this value. This means that the execution time of the cache manager algorithm does not depend on the entire dataset size, but rather only on the size of the managed cache. In our evaluation so far, the average execution time of the algorithm was 5 ms.

An orthogonal issue to throughput is collaboration between caches. Nearby caches, such as Frankfurt and Dublin, could collaborate in order to make better use of their shared storage size. As a first step, Agar nodes could broadcast their contents



(a) Vary cache size.

(b) Vary workload.

Fig. 8. Agar vs. different caching systems and the backend.

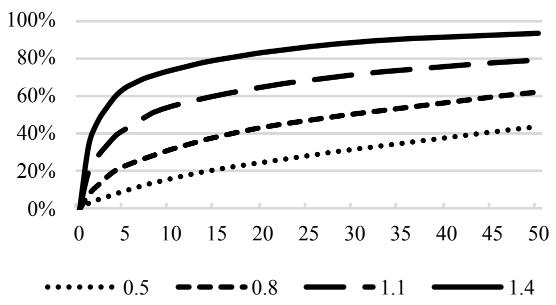


Fig. 9. Cumulative distribution of the object popularity using Zipfian workloads with different skews. The $y$ axis shows the cumulative percentage of requests in the workload that refer to objects on the $x$ axis (e.g., $x = 5$, $y = 40\%$ means that the most popular 5 objects account for 40% of requests).

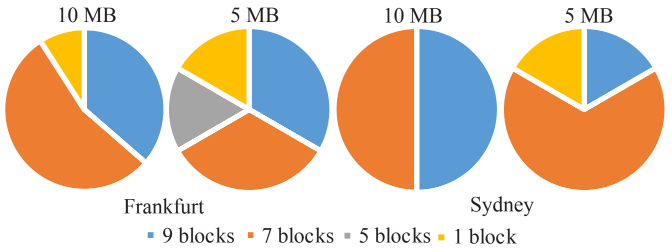


Fig. 10. Cache contents in different scenarios

and workload statistics periodically, in order to let nearby caches update the values of each cache option accordingly.

Finally, we also envision supporting data writes in our caching system. To allow this, Agar would need to implement a cache coherence algorithm, similar to CPUs. Protocols such as Paxos [16] could provide the necessary synchronization primitives to implement cache coherence.

## VII. Related Work

We first review in this section some of the classical caching policies proposed in the literature, and then focus on closely related work on caching applied in the context of erasure-coded data.

### A. Caching Policies

A caching policy represents a heuristic that is applied at a local caching node to pick an eviction candidate, when the cache is full. Jin et al. [17] identify three categories of caching policies, based on different properties of Internet-specific workloads: 1) temporal access locality, 2) access frequency, 3) a mix between the previous two. We extend this classification with a fourth category, as proposed by [18], that takes into account the size of objects.

*Least Recently Used (LRU):* LRU is a policy that relies on temporal access locality, which assumes that an object that has been recently requested is likely to be requested again in the near future. Thus, when the cache is full, LRU chooses the least recently accessed object as eviction candidate. The main advantage of LRU is that it automatically adapts to changes in access patterns. For example, Mokhtarian et al. [19] propose a LRU-based solution for caching in planet-scale video CDNs. While LRU is simpler to implement, Agar can bring better latency improvements.

*Least Frequently Used (LFU):* LFU is a policy that relies on metadata that captures the object access history: objects that are most popular are kept in the cache, at the expense of the less popular objects. LFU works best when the access pattern does not change much over time. The main challenge for LFU is to minimize the amount of metadata needed and still take good decisions regarding what objects to cache. A recent example of LFU-based policy is TinyLFU [15], a frequency-based cache admission policy: rather than deciding which object to evict, it decides whether it is worth admitting an object in the cache at the expense of the eviction candidate. Like Agar, TinyLFU is dedicated to caches subjected to skewed access distributions. TinyLFU builds upon Bloom filter theory and maintains an approximation of statistics regarding the access frequency for objects that have been requested recently. Unlike TinyLFU, Agar is designed for erasure coded data and tries to optimize the entire cache configuration rather than taking decisions per object. However, we believe that Agar can benefit from some of the optimizations in TinyLFU to make it more scalable.

*Hybrid LRU/LFU:* Some policies aim to combine the advantages of LRU and LFU by taking into account both the popularity and temporal locality of access in order to determine an optimal cache configuration. For example, WLFU [20] takes decisions based on statistics from the $W$ most recent requests, rather than keeping track of the entire object access history. WLFU uses LFU by default to choose which object to evict from the cache; if there are multiple objects with the same popularity score, WLFU uses LRU to break the tie and evict the least recently accessed object.

*Largest File First (LFF):* Since objects on the Web vary dramatically in size, some papers advocate for the idea of extending traditional caching policies to take into consideration the object size. GreedyDual-Size [21] combines recency of reference with object size and retrieval cost. An object is assigned an initial value based on its size and retrieval cost; this value is updated when the object is accessed again. Unlike Agar, GreedyDual-Size does not take the popularity of an object into account. GDSF [22] is a popularity-based extension to GreedyDual-Size. Like GreedyDual-Size, it computes the cost of objects based on information regarding the recency of access, and the size of an object, but also takes into account access frequency. In GDSF, larger objects have higher cost, and are, thus, more likely to be chosen as eviction candidates. LRU-SP [23] is a LRU extension that takes into account both the size and the popularity of an object. It is based on Size-Adjusted LRU [24]—a generalization of LRU that sorts cached objects in terms of the ratio between cost and size, and uses a greedy approach to evict those with the least cost-to-size ratio from the cache, and Segmented LRU [25]—a caching strategy designed to improve disk performance by assigning objects with different access frequency to different LRU queues. LRV [26] also takes into account the size, recency, and frequency of objects, but it is known for its large number of parameters and implementation overhead. While Agar draws inspiration from these approaches, none of these address the problems brought by keeping data coded. For example, we found that greedy algorithms are not suitable choices among Agar's caching options (§VI).

### B. Caching Erasure-Coded Data

CAROM [2] is a LRU-based caching scheme tailored for erasure-coded cloud file systems, whose workloads are known to exhibit temporal locality. CAROM considers both read and write operation and needs to provide consistency. CAROM totally orders writes by assigning each object to a primary data center, which becomes solely responsible for the object (encoding it and distributing the chunks during writes and storing the chunks during reads). CAROM mainly addresses the problem of supporting writes in erasure-coded systems, while Agar addresses the problem of optimizing cache configuration during reads.

Concurrent to our work, Aggarwal et al. [27] developed Sprout to address erasure coded chunks in caches. They develop an analytical model for the latencies of retrieving data and solve the integer optimization problem to find the cache parameters (cache contents) that minimize the latency. Agar differs in the approach to the problem, by mapping it to Knapsack. In the end, both Sprout and Agar obtain approximate solutions to the problem, as solving the optimization accurately is computationally intensive and impractical in a large system. While Sprout is still at the simulation level (at the time of this submission), we have deployed and evaluated the Agar prototype across a wide area network in Amazon Web Services. We are looking forward to further experimental validation of Sprout in order to draw conclusions on how Agar's strategy compares.

Rashmi et al. recently proposed EC-Cache [28], which applies online erasure-coding to objects stored in cluster caches. In contrast, Agar is a stand-alone caching system that augments multi-site erasure-coded storage systems with caches which it populates based on live information regarding data popularity and access latency to different storage sites.

## VIII. Conclusion

In this paper, we argued for the need for a caching system specifically developed for erasure-coded data, providing high availability with low latency and without the need to store full object replicas. We designed and implemented Agar, a caching system tailored for erasure-coded data, and explained how it integrates with a typical storage system. Agar uses a dynamic programming approach inspired by the Knapsack problem to optimize the cache configuration under a given workload. We compared our prototype with the LFU and LRU strategies and showed that Agar consistently outperforms them, obtaining 16%–41% lower latency.

## References

[1] H. Weatherspoon and J. Kubiatowicz, "Erasure coding vs. replication: A quantitative comparison," in *IPTPS*. Springer-Verlag, 2002.

[2] Y. Ma, T. Nandagopal, K. P. N. Puttaswamy, and S. Banerjee, "An ensemble of replication and erasure codes for cloud file systems," in *INFOCOM*. IEEE, 2013.

[3] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin, "Erasure Coding in Windows Azure Storage," in *ATC*. USENIX, 2012.

[4] M. Xia, M. Saxena, M. Blaum, and D. A. Pease, "A Tale of Two Erasure Codes in hdfs," in *FAST*. USENIX Association, 2015.

[5] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web caching and Zipf-like distributions: Evidence and implications," in *INFOCOM*. IEEE, 1999.

[6] F. Figueiredo, F. Benevenuto, and J. M. Almeida, "The tube over time: Characterizing popularity growth of YouTube videos," in *WSDM*. ACM, 2011.

[7] G. Szabo and B. A. Huberman, "Predicting the popularity of online content," *Communications of the ACM*, vol. 53, no. 8, 2010.

[8] G. Ananthanarayanan, A. Ghodsi, A. Warfield, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica, "PACMan: Coordinated Memory Caching for Parallel Jobs," in *NSDI 12*. USENIX, 2012.

[9] B. Fitzpatrick, "Distributed caching with Memcached," *Linux Journal*, vol. 2004, no. 124.

[10] "GitHub repository of the Yahoo! Cloud Serving Benchmark," https://github.com/brianfrankcooper/YCSB, Accessed: 2016-12-01.

[11] "GitHub repository of Longhair – Fast Cauchy Reed-Solomon Erasure Codes in C," https://github.com/catid/longhair, Accessed: 2016-12-01.

[12] "Knapscak problem," https://en.wikipedia.org/wiki/Knapsack_problem, Accessed: 2016-12-01.

[13] M. Kedia, "Lecture on Knapsack," https://www.cs.cmu.edu/afs/cs/academic/class/15854-f05/www/scribe/lec10.pdf, Accessed: 2016-12-01.

[14] "Fractional Knapsack," http://www.geeksforgeeks.org/fractional-knapsack-problem, Accessed: 2016-12-01.

[15] G. Einziger and R. Friedman, "TinyLFU: A highly efficient cache admission policy," in *PDP*. IEEE, 2014.

[16] L. Lamport, "Paxos Made Simple," *SIGACT News*, 2001.

[17] S. Jin and A. Bestavros, "GreedyDual*: Web caching algorithms exploiting the two sources of temporal locality in web request streams," in *WCW*, 2000.

[18] G. D. S. Silvestre, "Designing adaptive replication schemes for efficient content delivery in edge networks," Ph.D. dissertation, Universite Pierre et Marie Curie, 2013.

[19] K. Mokhtarian and H.-A. Jacobsen, "Caching in video CDNs: Building strong lines of defense," in *EuroSys*. ACM, 2014.

[20] G. Karakostas and D. Serpanos, "Exploitation of different types of locality for Web caches," in *ISCC*. IEEE, 2002.

[21] P. Cao and S. Irani, "Cost-aware WWW proxy caching algorithms," in *USITS*. USENIX, 1997.

[22] L. Cherkasova, "Improving WWW proxies performance with Greedy-Dual-Size-Frequency caching policy," HP Technical Report, Tech. Rep., 1998.

[23] K. Cheng and Y. Kambayashi, "LRU-SP: A size-adjusted and popularity-aware LRU replacement algorithm for Web caching," in *COMPSAC*, 2000.

[24] C. Aggarwal, J. L. Wolf, and P. S. Yu, "Caching on the World Wide Web," *Transactions on Knowledge and Data Engineering*, vol. 11, no. 1, 1999.

[25] R. Karedla, S. J. Love, and B. G. Wherry, "Caching strategies to improve disk system performance," *Computer*, vol. 27, no. 3, 1994.

[26] L. Rizzo and L. Vicisano, "Replacement policies for a proxy cache," *Journal IEEE/ACM TON*, vol. 8, no. 2, 2000.

[27] V. Aggarwal, Y.-F. R. Chen, T. Lan, and Y. Xiang, "Sprout: A functional caching approach to minimize service latency in erasure-coded storage," in *Poster session at the ICDCS*. IEEE, 2016.

[28] K. V. Rashmi, M. Chowdhury, J. Kosaian, I. Stoica, and K. Ramchandran, "EC-Cache: Load-Balanced, Low-Latency Cluster Caching with Online Erasure Coding," in *OSDI*. USENIX Association, 2016.