EC-Cache: Load-balanced, Low-latency Cluster Caching with Online Erasure Coding

K. V. Rashmi[1], Mosharaf Chowdhury[2], Jack Kosaian[2], Ion Stoica[1], Kannan Ramchandran[1]
[1]*UC Berkeley* [2]*University of Michigan*

Abstract

Data-intensive clusters and object stores are increasingly relying on in-memory object caching to meet the I/O performance demands. These systems routinely face the challenges of popularity skew, background load imbalance, and server failures, which result in severe load imbalance across storage servers and degraded I/O performance. Selective replication is a commonly used technique to tackle these challenges, where the number of cached replicas of an object is proportional to its popularity. In this paper, we explore an alternative approach using erasure coding.

EC-Cache is a load-balanced, low latency cluster cache that uses online erasure coding to overcome the limitations of selective replication. EC-Cache employs erasure coding by: (i) splitting and erasure coding individual objects during writes, and (ii) late binding, wherein obtaining any k out of $(k + r)$ splits of an object are sufficient, during reads. As compared to selective replication, EC-Cache improves load balancing by more than $3\times$ and reduces the median and tail read latencies by more than $2\times$, while using the same amount of memory. EC-Cache does so using 10% additional bandwidth and a small increase in the amount of stored metadata. The benefits offered by EC-Cache are further amplified in the presence of background network load imbalance and server failures.

1 Introduction

In recent years, in-memory solutions [12, 25, 56, 87, 89] have gradually replaced disk-based solutions [3, 29, 37] as the primary toolchain for high-performance data analytics. The root cause behind the transition is simple: in-memory I/O is orders of magnitude faster than that involving disks. Since the total amount of memory is significantly smaller than that of disks, the design of in-memory solutions boils down to maximizing the number of requests that can be efficiently served from memory. The primary challenges in this regard are: (i) Judiciously determining which data items to cache and which ones to evict. This issue has been well studied in past work [23, 35, 56, 67]. (ii) Increasing the *effective* memory capacity to be able to cache more data. Sampling [12, 16, 52] and compression [15, 27, 53, 79] are some of the popular approaches employed to increase the effective memory capacity. (iii) Ensuring good I/O performance for the cached data in the presence of skewed popularity, background load imbalance, and failures.

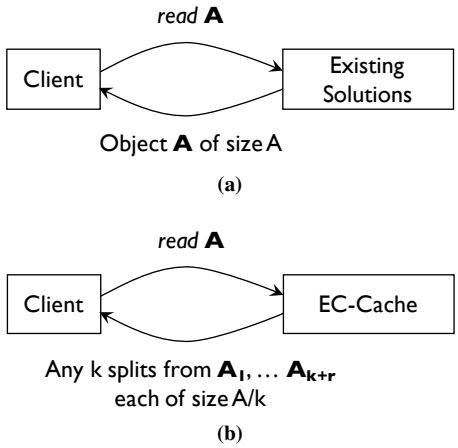


Figure 1: EC-Cache splits individual objects and encodes them using an erasure code to enable read parallelism and late binding during individual reads.

Typically, the popularity of objects in cluster caches are heavily skewed [20, 47], and this creates significant load imbalance across the storage servers in the cluster [20, 48]. The load imbalance necessitates overprovisioning in order to accommodate the peaks in the load distribution, and it also adversely affects the I/O performance. Consequently, load imbalance is one of the key challenges toward improving the performance of cluster caches. In addition to the skew in popularity, massive load fluctuations in the infrastructure due to background activities [33] and failures [70] can result in severe performance degradation.

A popular approach employed to address the aforementioned challenges is selective replication, which replicates objects based on their popularity [20, 63]; that is, it creates more replicas for hot objects. However, due to the limited amount of available memory, selective replication falls short in practice in terms of both load balancing and I/O performance [48].

While typical caches used in web-services and key-value stores cache small-sized objects in the range of a few bytes to few kilobytes, data-intensive cluster caches used for data analytics [23, 56] must store larger objects in the range of tens to hundreds of megabytes (§3). This significant increase in object sizes allows us to take a novel approach, using *erasure coding*, toward load balancing and improving I/O performance in cluster caches.

We present EC-Cache, an in-memory object cache that leverages *online erasure coding* – that is, data is never stored in a decoded form – to provide better load balancing and I/O performance (§4). We show both analytically (§5) and via extensive system evaluation (§6) that EC-Cache can outperform the optimal selective replication mechanism while using the same amount of memory.

EC-Cache employs erasure coding and its properties toward load balancing and improving I/O performance in the following manner.

Self-Coding and Load Spreading: A $(k,\ r)$ erasure code encodes k data units and generates r parity units such that any k of the $(k + r)$ total units are sufficient to decode the original k data units.[1] Erasure coding is traditionally employed in disk-based systems to provide fault-tolerance in a storage-efficient manner. In many such systems [26, 46, 61, 69], erasure coding is applied *across objects*: k objects are encoded to generate r additional objects. Read requests to an object are served from the original object unless it is missing. If the object is missing, it is reconstructed using the parities. In such a configuration, reconstruction using parities incurs huge bandwidth overheads [70]; hence, coding across objects is not useful for load balancing or improving I/O performance. In contrast, EC-Cache divides *individual* objects into k *splits* and creates r additional parity splits. Read requests to an object are served by reading any k of the $(k+r)$ splits and decoding them to recover the desired object (Figure 1). This approach provides multiple benefits. First, spreading the load of read requests across both data and parity splits results in better load balancing under skewed popularity. Second, reading/writing in parallel from multiple splits provides better I/O performance. Third, decoding the object using the parities does not incur any additional bandwidth overhead.

Late Binding: Under self-coding, an object can be reconstructed from *any* k of its $(k+r)$ splits. This allows us to leverage the power of choices: instead of reading exactly k splits, we read $(k + \Delta)$ splits (where $\Delta \leq r$) and wait for the reading of *any* k splits to complete. This late binding makes EC-Cache resilient to background load imbalance and unforeseen stragglers that are common in large clusters [24, 91], and it plays a critical role in taming tail latencies. Note that, while employing object splitting (that is, dividing each object into splits) together with selective replication can provide the benefits of load balancing and opportunities for read parallelism, this approach cannot exploit late binding without incurring high memory and bandwidth overheads (§ 2.3).

We have implemented EC-Cache over Alluxio [56] using Intel's ISA-L library [9]. It can be used as a caching layer on top of object stores such as Amazon S3 [2], Windows Azure Storage [30], and OpenStack Swift [11] where compute and storage are not collocated. It can also be used in front of cluster file systems such as HDFS [29], GFS [42], and Cosmos [31] by considering each block of a distributed file as an individual object.

We evaluated EC-Cache by deploying it on Amazon EC2 using synthetic workloads and production workload traces from a 3000-machine cluster at Facebook. EC-Cache improves the median and tail latencies for reads by more than $2\times$ in comparison to the optimal selective replication scheme; it improves load balancing by more than $3\times$,[2] while using the same amount of memory. EC-Cache's latency reductions increase as objects grow larger: for example, $1.33\times$ for 1 MB objects and $5.5\times$ for 100 MB objects. We note that using $k = 10$ and $\Delta = 1$ suffices to avail these benefits. In other words, a bandwidth overhead of at most 10% can lead to more than 50% reduction in the median and tail latencies. EC-Cache outperforms selective replication by even higher margins in the presence of an imbalance in the background network load and in the presence of server failures. Finally, EC-Cache performs well over a wide range of parameter settings.

Despite its effectiveness, our current implementation of EC-Cache offers advantages only for objects greater than 1 MB due to the overhead of creating $(k + \Delta)$ parallel TCP connections for each read. However, small objects form a negligible fraction of the footprint in many data-intensive workloads (§3). Consequently, EC-Cache simply uses selective replication for objects smaller than this threshold to minimize the overhead. Furthermore, EC-Cache primarily targets immutable objects, which is a popular model in many data analytics systems and object stores. Workloads with frequent, in-place updates are not suitable for EC-Cache because they would require updating all the parity splits of the updated objects.

Finally, we note that erasure codes are gaining increasing popularity in disk-based storage systems for providing fault tolerance in a space-efficient manner [26, 46, 61, 69]. EC-Cache demonstrates the effectiveness of erasure coding for a new setting – in-memory object caching – and toward new goals – improving load balancing and latency characteristics.

[1] Not all erasure codes have this property, but for simplicity, we do not make this distinction.

[2] This evaluation is in terms of the percent imbalance metric described in Section 6.

2 Background and Motivation

This section provides a brief overview of object stores (e.g., Amazon S3 [2], Windows Azure Storage [30], OpenStack Swift [11], and Ceph [86]) and in-memory caching solutions (e.g., Tachyon/Alluxio [56]) used in modern data-intensive clusters. We discuss the tradeoffs and challenges faced therein, followed by the opportunities for improvements over the state-of-the-art.

2.1 Cluster Caching for Object Stores

Cloud object stores [2, 11, 30, 86] provide a simple PUT/GET interface to store and retrieve arbitrary objects at an attractive price point. In recent years, due to the rapid increase in datacenter bandwidth [4, 77], cloud tenants are increasingly relying on these object stores as their primary storage solutions instead of compute-collocated cluster file systems such as HDFS [29]. For example, Netflix has been exclusively using Amazon S3 since 2013 [7]. Separating storage from compute in this manner mitigates disk locality challenges [62]. However, existing object stores can rarely offer end-to-end non-blocking connectivity without storage-side disks becoming a bottleneck. As a result, in-memory storage systems [56] are often used for caching in the compute side.

EC-Cache primarily targets storage-side caching to provide high I/O performance while mitigating the need for compute-side caching. Note that, in the presence of very high-speed networks, it can also be used in environments where compute and storage are collocated.

2.2 Challenges in Object Caching

In-memory object caches face unique tradeoffs and challenges due to workload variations and dynamic infrastructure in large-scale deployments.

Popularity Skew Recent studies from production clusters show that the popularity of objects in cluster caches are heavily skewed [20, 47], which creates significant load imbalance across storage servers in the cluster. This hurts I/O performance and also requires over-provisioning the cluster to accommodate the peaks in the load distribution. Unsurprisingly, load imbalance has been reported to be one of the key challenges toward improving the performance of cluster caches [45, 48].

Background Load Imbalance Across the Infrastructure In addition to skews in object popularity, network interfaces – and I/O subsystems in general – throughout the cluster experience massive load fluctuations due to background activities [33]. Predicting and reacting to these variations in time is difficult. Even with selective replication, performance can deteriorate significantly if the source of an object suddenly becomes a hotspot (§6.3).

Tradeoff Between Memory Efficiency, Fault Tolerance, and I/O Performance In caches, fault tolerance and I/O performance are inherently tied together since failures result in disk I/O activities, which, in turn, significantly increases latency. Given that memory is a constrained and expensive resource, existing solutions either sacrifice fault tolerance (that is, no redundancy) to increase memory efficiency [56, 89], or incur high memory overheads (e.g., replication) to provide fault tolerance [81, 90].

2.3 Potential for Benefits

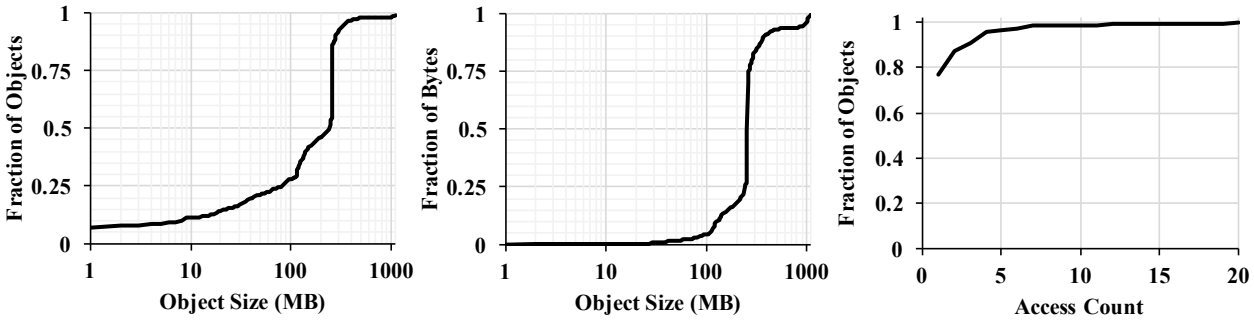
Due to the challenges of popularity skew, background load imbalance, and failures, maintaining a single copy of each object in memory is often insufficient for acheiving high performance. Replication schemes that treat all objects alike do not perform well under popularity skew as they waste memory by replicating less-popular objects. Selective replication [20, 45, 63], where additional replicas of hot objects are cached, only provides coarse-grained support: each replica incurs an additional memory overhead of $1\times$. Selective replication has been shown to fall short in terms of both load balancing and I/O performance [48] (§6.2).

Selective replication along with object splitting (all splits of the same object have the same replication factor) does not solve the problem either. While such an object-splitting approach provides better load balancing and opportunities for read parallelism, it cannot exploit late binding without incurring high memory and bandwidth overheads. As shown in Section 6.6.2, contacting multiple servers to read the splits severely affects tail latencies, and late binding is necessary to rein them in. Hence, under selective replication with object splitting, each object will need at least $2\times$ memory overhead, and, in order to make use of late binding, one must read multiple copies of each of the splits of the object, resulting in at least $2\times$ bandwidth overhead.

3 Analysis of Production Workload

Object stores are gaining popularity as the primary data storage solution for data analytics pipelines (e.g., at Netflix [6, 7]). As EC-Cache is designed to cater to these use cases, in order to obtain a better understanding of the requirements, we analyzed a trace with millions of reads in a 3000-machine analytics cluster at Facebook. The trace was collected in October 2010, and consists of a mix of batch and interactive MapReduce analytics jobs generated from Hive queries. The block size for the HDFS installation in this cluster was 256 MB, and the corresponding network had a $10:1$ oversubscription ratio.

Our goal in analyzing these traces is to highlight characteristics – distributions of object sizes, their relative



(a) Object size **(b)** Object footprint **(c)** Object access characteristics [23]

Figure 2: Characteristics of object reads in the Facebook data analytics cluster. We observe that (a) large object sizes are more prevalent; (b) small objects have even smaller footprint; and (c) access patterns across objects is heavily skewed. Note that the X-axes are in log-scale in (a) and (b).

impact, access characteristics, and the nature of imbalance in I/O utilizations – that enable us to make realistic assumptions in our analysis, design, and evaluation.

3.1 Large Object Reads are Prevalent

Data-intensive jobs in production clusters are known to follow skewed distributions in terms of their input and output size [23, 32, 34]. We observe a similar skewed pattern in the Facebook trace (Figure 2): only 7% (11%) of the reads are smaller than 1 (10) MB, but their total size in terms of storage usage is miniscule. Furthermore, 28% of the objects are less than 100 MB in size with less than 5% storage footprint. Note that a large fraction of the blocks in the Facebook cluster are 256 MB in size, which corresponds to the vertical segment in Figure 2a.

3.2 Popularity of Objects is Skewed

Next, we focus on object popularity/access patterns. As noted in prior work [20, 23, 32, 56, 61], object popularity follows a Zipf-like skewed pattern; that is, a small fraction of the objects are highly popular. Figure 2c [23, Figure 9] plots the object access characteristics. Note that this measurement does not include objects that were never accessed. Here, the most popular 5% of the objects are seven times more popular than the bottom three-quarters [23].

3.3 Network Load Imbalance is Inherent

As observed in prior studies [28, 33, 44, 51], we found that datacenter traffic across the oversubscribed links can be significantly imbalanced. Furthermore, network imbalances are time varying. The root causes behind such imbalances include, among others, skew in application-level communication patterns [28, 51, 55], rolling upgrades and maintenance operations [28], and imperfect load balancing inside multipath datacenter networks [19]. We measured the network imbalance as the ratio of the maximum and the average utilizations across all

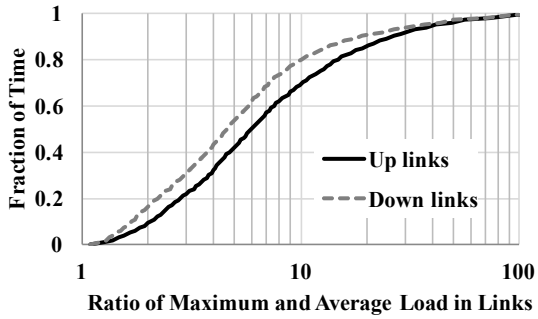


Figure 3: Imbalance in utilizations (averaged over 10-second intervals) of up and down oversubscribed links in Facebook clusters due to data analytics workloads. The X-axis is in log-scale.

oversubscribed links[3] in the Facebook cluster (Figure 3). This ratio was above 4.5× more than 50% of the time for both up and downlinks, indicating significant imbalance. Moreover, the maximum utilization was high for a large fraction of the time, thereby increasing the possibility of congestion. For instance, the maximum uplink utilization was more than 50% of the capacity for more than 50% of the time. Since operations on object stores must go over the network, network hotspots can significantly impact their performance. This impact is amplified for in-memory object caches, where the network is the primary bottleneck.

4 EC-Cache Design Overview

This section provides a high-level overview of EC-Cache's architecture.

4.1 Overall Architecture

EC-Cache is an object caching solution to provide high I/O performance in the presence of popularity skew, background load imbalance, and server failures. It con-

[3] Links connecting top-of-the-rack (ToR) switches to the core.

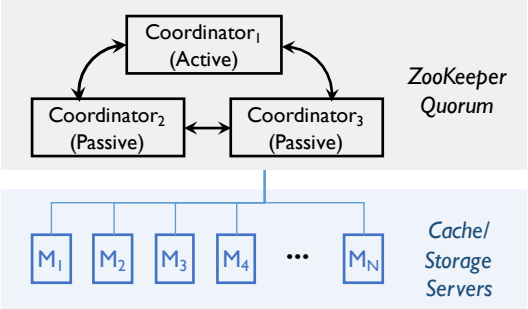
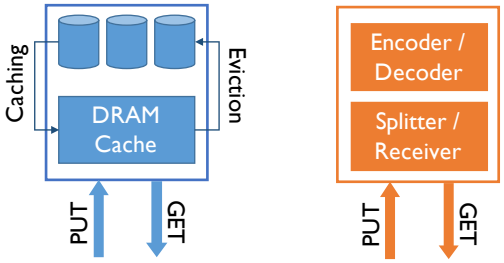


Figure 4: Alluxio architecture



(a) Backend server **(b)** EC-Cache client

Figure 5: Roles in EC-Cache: (a) backend servers manage interactions between caches and persistent storage for client libraries; (b) EC-Cache clients perform encoding and decoding during writes and reads.

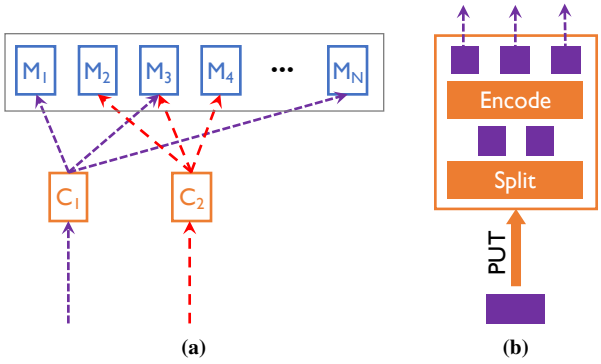
sists of a set of servers, each of which has an in-memory cache on top of on-disk storage. Applications interact with EC-Cache via a client library. Similar to other object stores [2, 11, 30], EC-Cache storage servers are not collocated with the applications using EC-Cache.

We have implemented EC-Cache on top of Alluxio [56], which is a popular caching solution for big data clusters. Consequently, EC-Cache shares some high-level similarities with Alluxio's architecture, such as a centralized architecture with a master coordinating several storage/cache servers (Figure 4).

Backend Storage Servers Both in-memory and on-disk storage in each server is managed by a worker that responds to read and write requests for splits from clients (Figure 5a). Backend servers are unaware of object-level erasure coding introduced by EC-Cache. They also take care of caching and eviction of objects to and from memory using the least-recently-used (LRU) heuristic [56].

EC-Cache Client Library Applications use EC-Cache through a PUT-GET interface (Figure 5b). The client library transparently handles all aspects of erasure coding.

EC-Cache departs significantly from Alluxio in two major ways in its design of the user-facing client library. First, EC-Cache's client library exposes a significantly narrower interface for object-level operations as



(a) **(b)**

Figure 6: Writes to EC-Cache. (a) Two concurrent writes with $k = 2$ and $r = 1$ from two applications. (b) Steps involved during an individual write inside the EC-Cache client library for an object using $k = 2$ and $r = 1$.

compared to Alluxio's file-level interface. Second, EC-Cache's client library takes care of splitting and encoding during writes, and reading from splits and decoding during reads instead of writing and reading entire objects.

4.2 Writes

EC-Cache stores each object by dividing it into k splits and encoding these splits using a Reed-Solomon code [73] to add r parity splits.[4] It then distributes these $(k + r)$ splits across unique backend servers chosen uniformly at random. Note that each object is allowed to have distinct values of the parameters k and r. Figure 6 depicts an example of object writes with $k = 2$ and $r = 1$ for both objects C_1 and C_2. EC-Cache uses Intel ISA-L [9] for encoding operations.

A key issue in any distributed storage solution is that of data placement and rendezvous, that is, where to write and where to read from. The fact that each object is further divided into $(k+r)$ splits in EC-Cache magnifies this issue. For the same reason, metadata management is also an important issue in our design. Similar to Alluxio and most other storage systems [29, 42, 56], the EC-Cache coordinator determines and manages the locations of all the splits. Each write is preceded by an interaction with the coordinator server that determines where each of the $(k + r)$ splits are to be written. Similarly, each reader receives the locations of the splits through a single interaction with the coordinator.

EC-Cache requires a minimal amount of additional metadata to support object splitting. For each object, EC-Cache stores its associated k and r values and the associated $(k + r)$ server locations (32-bit unsigned integers). This forms only a small fraction of the total metadata size of an object.

[4] Section 4.4 discusses the choice of the erasure coding scheme.

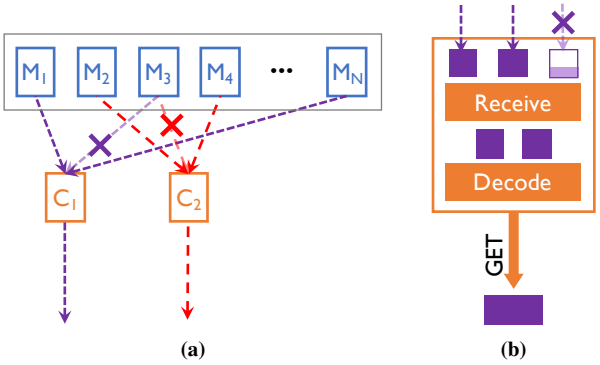


Figure 7: Reads from EC-Cache. (a) Two concurrent reads with $k = 2$ and $r = 1$. Reads from M_3 are slow and hence are ignored (crossed). (b) Steps involved in an individual read in the client library for an object with $k = 2, r = 1$, and $\Delta = 1$.

4.3 Reads

The key advantage of EC-Cache comes into picture during read operations. Instead of reading from a single replica, the EC-Cache client library reads from $(k + \Delta)$ splits in parallel chosen uniformly at random (out of the $(k + r)$ total splits of the object). This provides three benefits. First, it exploits I/O parallelism. Second, it distributes the load across many backend servers helping in balancing the load. Third, the read request can be completed as soon as any k out of $(k + \Delta)$ splits arrive, thereby avoiding stragglers. Once k splits of an object arrives, the decoding operation is performed using Intel ISA-L [9].

Figure 7a provides an example of a read operation over the objects stored in the example presented in Figure 6. In this example, both the objects have $k = 2$ and $r = 1$. Although 2 splits are enough to complete each read request, EC-Cache issues an additional read (that is, $\Delta = 1$). Since both objects had one split in server M_3, reading from that server may be slow. However, instead of waiting for that split, EC-Cache proceeds as soon as it receives the other 2 splits (Figure 7b) and decodes them to complete the object read requests.

Additional reads play a critical role in avoiding stragglers, and thus, in reducing tail latencies. However, they also introduce additional load on the system. The bandwidth overhead due to additional reads is precisely $\frac{\Delta}{k}$. In Section 6.6.2, we present a sensitivity analysis with respect to Δ, highlighting the interplay between the above two aspects.

4.4 Choice of Erasure Code

In disk-based storage systems, erasure codes are employed primarily to provide fault tolerance in a storage-efficient manner. In these systems, network and I/O resources consumed during recovery of failed or otherwise unavailable data units play a critical role in the choice of the erasure code employed [46, 69, 70]. There has been a considerable amount of recent work on designing erasure codes for distributed storage systems to optimize recovery operations [26, 43, 68, 71, 72]. Many distributed storage systems are adopting these recovery-optimized erasure codes in order to reduce network and I/O consumption [8, 46, 69]. On the other hand, EC-Cache employs erasure codes for load balancing and improving read performance of cached objects. Furthermore, in this caching application, recovery operations are not a concern as data is persisted in the underlying storage layer.

We have chosen to use Reed-Solomon (RS) [73] codes for two primary reasons. First, RS codes are Maximum-Distance-Separable (MDS) codes [59]; that is, they possess the property that any k out of the $(k + r)$ splits are sufficient to decode the object. This property provides maximum flexibility in the choice of splits for load balancing and late binding. Second, the Intel ISA-L [9] library provides a highly optimized implementation of RS codes that significantly decreases the time taken for encoding and decoding operations. This reduced decoding complexity makes it feasible for EC-Cache to perform decoding for every read operation. Both the above factors enable EC-Cache to exploit properties of erasure coding to achieve significant gains in load balancing and read performance (§6).

5 Analysis

In this section, we provide an analytical explanation for the benefits offered by EC-Cache.

5.1 Impact on Load Balancing

Consider a cluster with S servers and F objects. For simplicity, let us first assume that all objects are equally popular. Under selective replication, each object is placed on a server chosen uniformly at random out of the S servers. For simplicity, first consider that EC-Cache places each split of a object on a server chosen uniformly at random (neglecting the fact that each split is placed on a unique server). The total load on a server equals the sum of the loads on each of the splits stored on that server. Thus the load on each server is a random variable. Without loss of generality, let us consider the load on any particular server and denote the corresponding random variable by L.

The variance of L directly impacts the load imbalance in the cluster – intuitively, a higher variance of L implies a higher load on the maximally loaded server in comparison to the average load; consequently, a higher load imbalance.

Under this simplified setting, the following result holds.

Theorem 1 For the setting described above:

$$\frac{\text{Var}(L_{\text{EC-Cache}})}{\text{Var}(L_{\text{Selective Replication}})} = \frac{1}{k}.$$

Proof: Let $w > 0$ denote the popularity of each of the files. The random variable $L_{\text{Selective Replication}}$ is distributed as a Binomial random variable with F trials and success probability $\frac{1}{S}$, scaled by w. On the other hand, $L_{\text{EC-Cache}}$ is distributed as a Binomial random variable with kF trials and success probability $\frac{1}{S}$, scaled by $\frac{w}{k}$. Thus we have

$$\frac{\text{Var}(L_{\text{EC-Cache}})}{\text{Var}(L_{\text{Selective Replication}})} = \frac{\left(\frac{w}{k}\right)^2 (kF)\frac{1}{S}\left(1-\frac{1}{S}\right)}{w^2 F \frac{1}{S}\left(1-\frac{1}{S}\right)} = \frac{1}{k},$$

thereby proving our claim. □

Intuitively, the splitting action of EC-Cache leads to a smoother load distribution in comparison to selective replication. One can further extend Theorem 1 to accommodate a skew in the popularity of the objects. Such an extension leads to an identical result on the ratio of the variances. Additionally, the fact that each split of an object in EC-Cache is placed on a unique server further helps in evenly distributing the load, leading to even better load balancing.

5.2 Impact on Latency

Next, we focus on how object splitting impacts read latencies. Under selective replication, a read request for an object is served by reading the object from a server. We first consider naive EC-Cache without any additional reads. Under naive EC-Cache, a read request for an object is served by reading k of its splits in parallel from k servers and performing a decoding operation. Let us also assume that the time taken for decoding is negligible compared to the time taken to read the splits.

Intuitively, one may expect that reading splits in parallel from different servers will reduce read latencies due to the parallelism. While this reduction indeed occurs for the average/median latencies, the tail latencies behave in an opposite manner due to the presence of stragglers – one slow split read delays the completion of the entire read request.

In order to obtain a better understanding of the aforementioned phenomenon, let us consider the following simplified model. Consider a parameter $p \in [0, 1]$ and assume that for any request, a server becomes a straggler with probability p, independent of all else. There are two primary contributing factors to the distributions of the latencies under selective replication and EC-Cache:

(a) Proportion of stragglers: Under selective replication, the fraction of requests that hit stragglers is p. On the other hand, under EC-Cache, a read request for an object will face a straggler if any of the k servers from where splits are being read becomes a straggler. Hence, a higher fraction $\left(1-(1-p)^k\right)$ of read requests can hit stragglers under naive EC-Cache.

(b) Latency conditioned on absence/presence of stragglers: If a read request does not face stragglers, the time taken for serving a read request is significantly smaller under EC-Cache as compared to selective replication because splits can be read in parallel. On the other hand, in the presence of a straggler in the two scenarios, the time taken for reading under EC-Cache is about as large as that under selective replication.

Putting the aforementioned two factors together we get that the relatively higher likelihood of a straggler under EC-Cache increases the number of read requests incurring a higher latency. The read requests that do not encounter any straggler incur a lower latency as compared to selective replication. These two factors explain the decrease in the median and mean latencies, and the increase in the tail latencies.

In order to alleviate the impact on tail latencies, we use additional reads and late binding in EC-Cache. Reed-Solomon codes have the property that any k of the collection of all splits of an object suffice to decode the object. We exploit this property by reading more than k splits in parallel, and using the k splits that are read first. It is well known that such additional reads help in mitigating the straggler problem and alleviate the affect on tail latencies [36, 82].

6 Evaluation

We evaluated EC-Cache through a series of experiments on Amazon EC2 [1] clusters using synthetic workloads and traces from Facebook production clusters. The highlights of the evaluation results are:

- For skewed popularity distributions, EC-Cache improves load balancing over selective replication by $3.3\times$ while using the same amount of memory. EC-Cache also decreases the median latency by $2.64\times$ and the 99.9th percentile latency by $1.79\times$ (§6.2).
- For skewed popularity distributions *and* in the presence of background load imbalance, EC-Cache decreases the 99.9th percentile latency w.r.t. selective replication by $2.56\times$ while maintaining the same benefits in median latency and load balancing as in the case without background load imbalance (§6.3).
- For skewed popularity distributions *and* in the presence of server failures, EC-Cache provides a graceful degradation as opposed to the significant degradation in tail latency faced by selective replication. Specifically, EC-Cache decreases the 99.9th percentile latency w.r.t. selective replication by $2.8\times$ (§6.4).
- EC-Cache's improvements over selective replication increase as object sizes increase in production traces;

e.g., 5.5× at median for 100 MB objects with an upward trend (§6.5).

- EC-Cache outperforms selective replication across a wide range of values of k, r, and Δ (§6.6).

6.1 Methodology

Cluster Unless otherwise specified, our experiments use 55 `c4.8xlarge` EC2 instances. 25 of these machines act as the backend servers for EC-Cache, each with 8 GB cache space, and 30 machines generate thousands of read requests to EC-Cache. All machines were in the same Amazon Virtual Private Cloud (VPC) with 10 Gbps enhanced networking enabled; we observed around 4-5 Gbps bandwidth between machines in the VPC using `iperf`.

As mentioned earlier, we implemented EC-Cache on Alluxio [56], which, in turn, used Amazon S3 [2] as its persistence layer and runs on the 25 backend servers. We used DFS-Perf [5] to generate the workload on the 30 client machines.

Metrics Our primary metrics for comparison are *latency* in reading objects and *load imbalance* across the backend servers.

Given a workload, we consider mean, median, and high-percentile latencies. We measure improvements in latency as:

$$\text{Latency Improvement} = \frac{\text{Latency w/ Compared Scheme}}{\text{Latency w/ EC-Cache}}$$

If the value of this "latency improvement" is greater (or smaller) than one, EC-Cache is better (or worse).

We measure load imbalance using the percent imbalance metric λ defined as follows:

$$\lambda = \left(\frac{L_{\max} - L_{\text{avg}^\star}}{L_{\text{avg}^\star}}\right) * 100, \quad (1)$$

where $L_{\max}$ is the load on the server which is maximally loaded and $L_{\text{avg}^\star}$ is the load on any server under an *oracle* scheme, where the total load is equally distributed among all the servers without any overhead. λ measures the percentage of additional load on the maximally loaded server as compared to the ideal average load. Because EC-Cache operates in the bandwidth-limited regime, the load on a server translates to the total amount of data read from that server. Lower values of λ are better. Note that the percent imbalance metric takes into account the additional load introduced by EC-Cache due to additional reads.

Setup We consider a Zipf distribution for the popularity of objects, which is common in many real-world object popularity distributions [20, 23, 56]. Specifically, we consider the Zipf parameter to be 0.9 (that is, high skew).

Unless otherwise specified, we allow both selective replication and EC-Cache to use 15% memory overhead to handle the skew in the popularity of objects. Selective replication uses all the allowed memory overhead for handling popularity skew. Unless otherwise specified, EC-Cache uses $k = 10$ and $\Delta = 1$. Thus, 10% of the allowed memory overhead is used to provide one parity to each object. The remaining 5% is used for handling popularity skew. Both schemes make use of the skew information to decide how to allocate the allowed memory among different objects in an identical manner: the number of replicas for an object under selective replication and the number of additional parities for an object under EC-Cache are calculated so as to flatten out the popularity skew to the extent possible starting from the most popular object, until the memory budget is exhausted.

Moreover, both schemes use uniform random placement policy to evenly distribute objects (splits in case of EC-Cache) across memory servers.

Unless otherwise specified, the size of each object considered in these experiments is 40 MB. We present results for varying object sizes observed in the Facebook trace in Section 6.5. In Section 6.6, we perform a sensitivity analysis with respect to all the above parameters.

Furthermore, we note that while the evaluations presented here are for the setting of high skew in object popularity, EC-Cache outperforms selective replication in scenarios with low skew in object popularity as well. Under high skew, EC-Cache offers significant benefits in terms of load balancing and read latency. Under low skew, while there is not much to improve in load balancing, EC-Cache will still provide latency benefits.

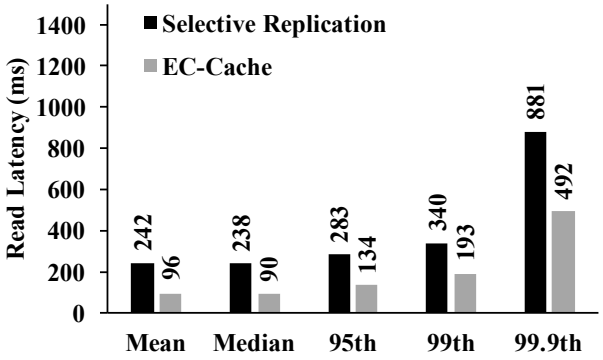
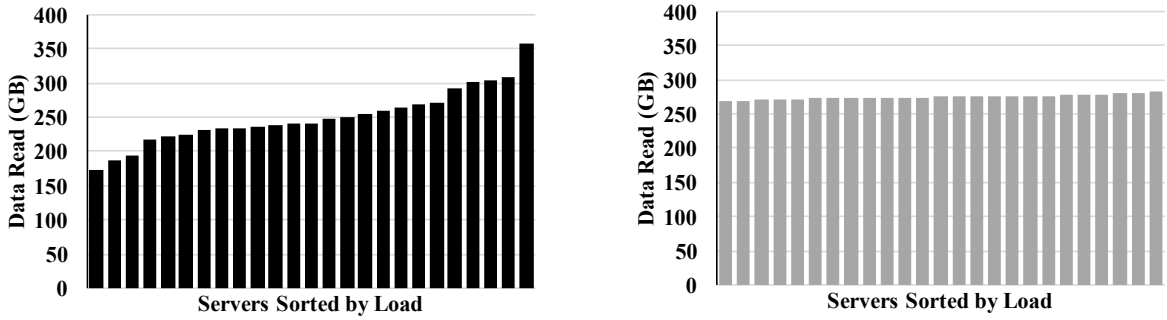


Figure 8: Read latencies under skewed popularity of objects.

6.2 Skew Resilience

We begin by evaluating the performance of EC-Cache in the presence of skew in object popularity.

Latency Characteristics Figure 8 compares the mean, median, and tail latencies of EC-Cache and selective replication. We observe that EC-Cache improves median and mean latencies by 2.64× and 2.52×, respectively. EC-Cache outperforms selective replication at high per-



(a) Selective replication

(b) EC-Cache with $k = 10$

Figure 9: Comparison of load distribution across servers in terms of the amount of data read from each server. The percent imbalance metric λ for selective replication and EC-Cache are 43.45% and 13.14% respectively.

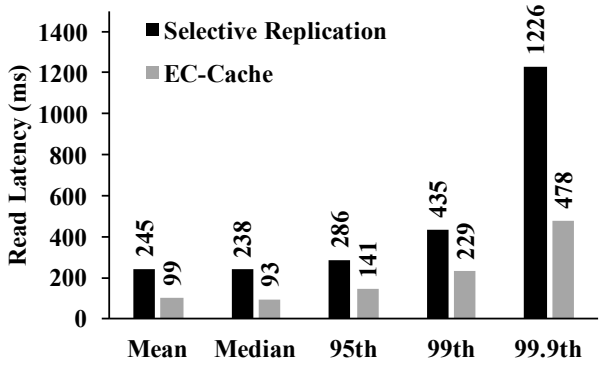


Figure 10: Read latencies in the presence of background traffic from big data workload.

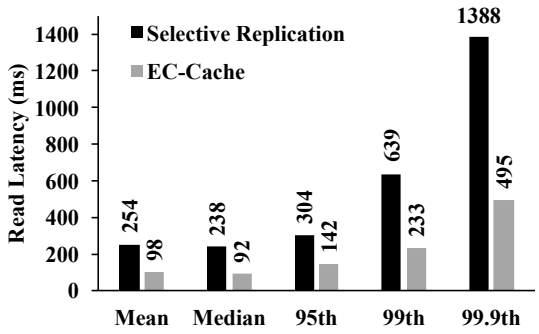


Figure 11: Read latencies in the presence of server failures.

centiles as well, improving the latency by $1.76\times$ at the 99th percentile and by $1.79\times$ at the 99.9th percentile.

Load Balancing Characteristics Figure 9 presents the distribution of loads across servers. The percent imbalance metric λ observed for selective replication and EC-Cache in this experiment are 43.45% and 13.14% respectively.

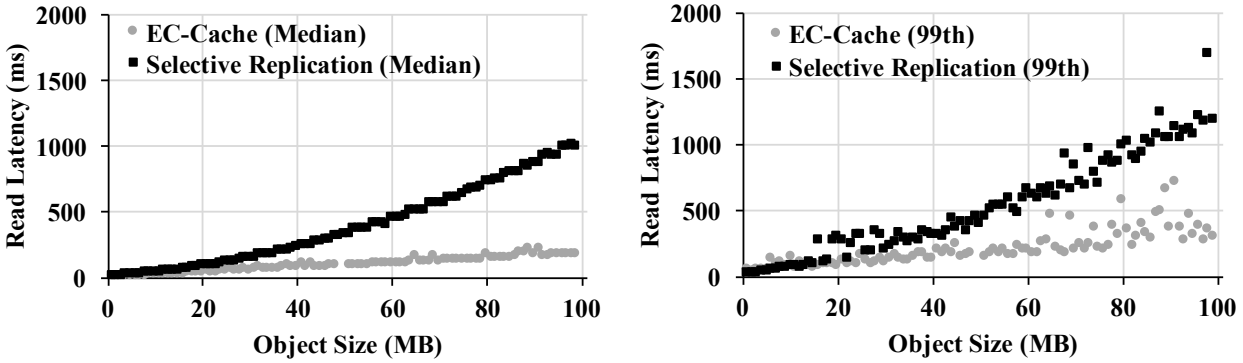
Decoding Overhead During Reads We observed that the time taken to decode during the reads is approximately 30% of the total time taken to complete a read request. Despite this overhead, we see (Figure 8) that EC-Cache provides a significant reduction in both median and tail latencies. Although our current implementation uses only a single thread for decoding, the underlying erasure codes permit the decoding process to be made embarrassingly parallel, potentially allowing for a linear speed up; this, in turn, can further improve EC-Cache's latency characteristics.

6.3 Impact of Background Load Imbalance

We now investigate EC-Cache's performance in the presence of a background network load, specifically in the presence of unbalanced background traffic. For this experiment, we generated a background load that follows traffic characteristics similar to those described in Section 3.3. Specifically, we emulated network transfers from shuffles for the jobs in the trace. Shuffles arrive following the same arrival pattern of the trace. For each shuffle, we start some senders (emulating mappers) and receivers (emulating reducers) that transfer randomly generated data over the network. The amount of data received by each receiver for each shuffle followed a distribution similar to that in the trace.

Latency Characteristics Figure 10 compares the mean, median, and tail latencies using both EC-Cache and selective replication. We observe that as in Section 6.2, EC-Cache improves the median and mean latencies by $2.56\times$ and $2.47\times$ respectively.

At higher percentiles, EC-Cache's benefits over selective replication are even more than that observed in Section 6.2. In particular, EC-Cache outperforms selective replication by $1.9\times$ at the 99th percentile and by $2.56\times$ at the 99.9th percentile. The reason for these improvements is the following: while selective replication gets stuck in few of the overloaded backend servers, EC-Cache remains almost impervious to such imbalance due to late binding.



(a) Median latency

(b) 99th percentile latency

Figure 12: Comparison of EC-Cache and selective replication read latencies over varying object sizes in the Facebook production trace. EC-Cache's advantages improve as objects become larger.

Load Balancing Characteristics The percent imbalance metric λ for selective replication and EC-Cache are similar to that reported in Section 6.2. This is because the imbalance in background load does not affect the load distribution across servers due to read requests.

6.4 Performance in Presence of Failures

We now evaluate the performance of EC-Cache in the presence of server failures. This experiment is identical to that in Section 6.2 except with one of the back-end servers terminated. The read latencies in this degraded mode are shown in Figure 11. Comparing the latencies in Figure 8 and Figure 11, we see that the performance of EC-Cache does not degrade much as most objects are still served from memory. On the other hand, selective replication suffers significant degradation in tail latencies as some of the objects are now served from the underlying storage system. Here, EC-Cache outperforms selective replication by $2.7\times$ at the 99th percentile and by $2.8\times$ at the 99.9th percentile.

6.5 Performance on Production Workload

So far we focused on EC-Cache's performance for a fixed object size. In this section, we compare EC-Cache against selective replication for varying object sizes based on the workload collected from Facebook (details in Section 3).

Figure 12 presents the median and the 99th percentile read latencies for objects of different sizes (starting from 1 MB). Note that EC-Cache resorts to selective replication for objects smaller than 1 MB to avoid communication overheads.

We make two primary observations. First, EC-Cache's median improvements over selective replication steadily increases with the object size; e.g., EC-Cache is $1.33\times$ faster for 1 MB-sized objects, which improves to $5.5\times$ for 100 MB-sized objects and beyond. Second, EC-Cache's 99th percentile improvements over selective replication kick off when object sizes grow beyond 10 MB. This is because EC-Cache's constant overhead of establishing $(k + \Delta)$ connections is more pronounced for smaller reads, which generally have lower latencies. Beyond 10 MB, connection overheads get amortized due to increased read latency, and EC-Cache's improvements over selective replication even in tail latencies steadily increase from $1.25\times$ to $3.85\times$ for 100 MB objects.

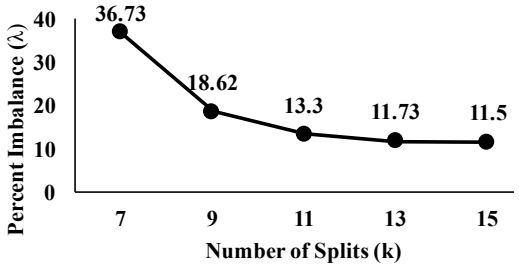


Figure 13: Load imbalance for varying values of k with $\Delta = 1$: percent imbalance metric (λ) decreases as objects are divided into more splits.

6.6 Sensitivity Evaluation

In this section, we evaluate the effects of the choice of different EC-Cache parameters. We present the results for 10 MB objects (instead of 40 MB as in prior evaluations) in order to bring out the effects of all the parameters more clearly and to be able to sweep for a wide range of parameters.

6.6.1 Number of splits k

Load Balancing Characteristics The percent imbalance metric for varying values of k with $\Delta = 1$ are shown in Figure 13. We observe that load balancing improves with increasing k. There are two reasons for this phenomenon: (i) A higher value of k leads to a smaller granularity of individual splits, thereby resulting in a

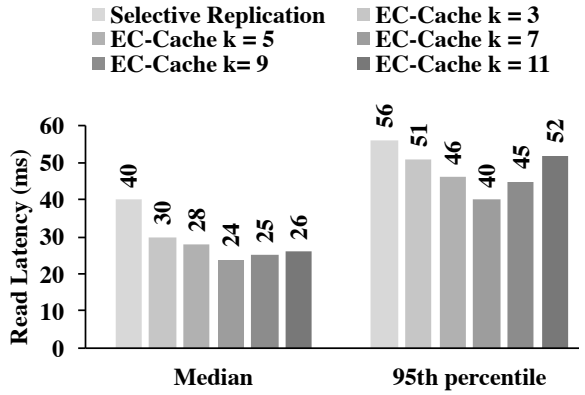


Figure 14: Impact of the number of splits k on the read latency.

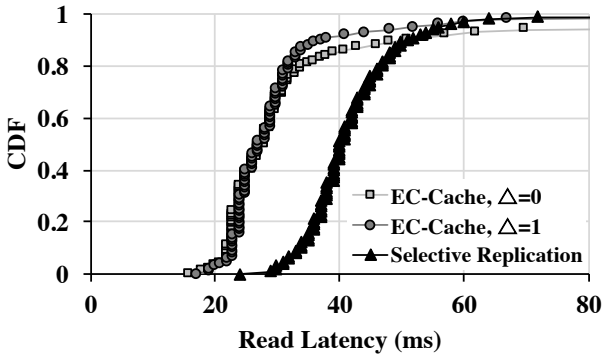


Figure 15: CDF of read latencies showing the need for additional reads in reining in tail latencies in EC-Cache.

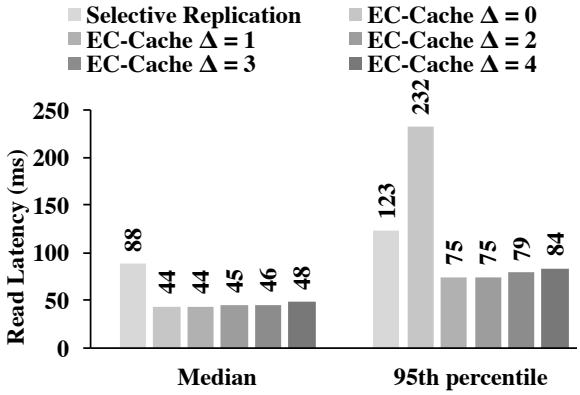


Figure 16: Impact of the number of additional reads on read latency.

greater smoothing of the load under skewed popularity. (ii) With a fixed value of Δ, the load overhead due to additional reads varies inversely with the value of k. This trend conforms to the theoretical analysis presented in Section 5.1.

Latency Characteristics Figure 14 shows a comparison of median and 95th percentile read latencies for varying values of k with $\Delta = 1$. The corresponding values for selective replication are also provided for comparison. We observe that parallelism helps in improving median latencies, but with diminishing returns. However, higher values of k lead to worse tail latencies as a result of the straggler effect discussed earlier in Section 5. Hence, for $k > 10$, more than one additional reads are needed to rein in the tail latencies. We elaborate this effect below.

6.6.2 Additional Reads (Δ)

First, we study the necessity of additional reads. Figure 15 shows the CDF of read latencies from about $160,000$ reads for selective replication and EC-Cache with $k = 10$ with and without additional reads, that is, with $\Delta = 1$ and $\Delta = 0$, respectively. We observe that, without any additional reads, EC-Cache performs quite well in terms of the median latency, but severely suffers at high percentiles. This is due to the effect of stragglers as discussed in Section 5.2. Moreover, adding just one additional read helps EC-Cache tame these negative effects. Figure 15 also shows that selective replication with object splitting (as discussed in Section 2.3) would not perform well.

Next, we study the effect of varying values of Δ. In this experiment, we vary Δ from 0 to 4, set $k = 12$, and use an object size of 20 MB. We choose $k = 12$ instead of 10 because the effect of additional reads is more pronounced for higher values of k, and we choose a larger object size (20 MB instead of 10 MB) because the value of k is higher (§7.4). We use uniform popularity distribution across objects so that each object is provided with equal (specifically, $r = 4$) number of parities. This allows us to evaluate with values of Δ up to 4. Figure 16 shows the impact of different number of additional reads on the read latency. We see that the first one or two additional reads provide a significant reduction in the tail latencies while subsequent additional reads provide little additional benefits. In general, having too many additional reads would start hurting the performance because they would cause a proportional increase in communication and bandwidth overheads.

6.6.3 Memory Overhead

Up until now, we have compared EC-Cache and selective replication with a fixed memory overhead of 15%. Given a fixed amount of total memory, increasing memory overhead allows a scheme to cache more redundant objects but fewer unique objects. In this section, we vary memory overhead and evaluate the latency and load balancing characterisitics of selective replication and EC-Cache.

We observed that the relative difference in terms of latency between EC-Cache and selective replication remained similar to that shown in Figure 8 – EC-Cache provided a significant reduction in the median and tail latencies as compared to selective replication even for higher memory overheads. However, in terms of load

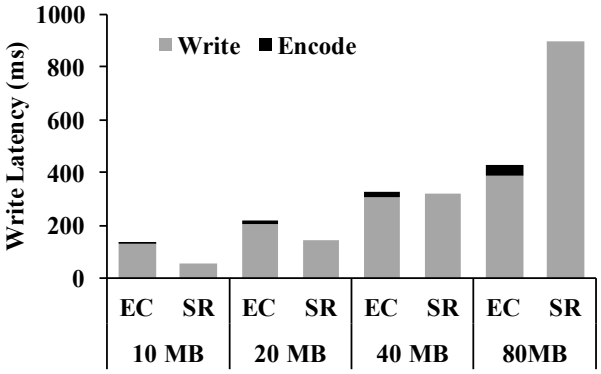


Figure 17: Comparison of writing times (and encoding time for EC-Cache) for different object sizes.

balancing, the gap between EC-Cache and selective replication decreased with increasing memory overhead. This is because EC-Cache was almost balanced even with just 15% memory overhead (Figure 9) with little room for further improvement. In contrast, selective replication became more balanced due to the higher memory overhead allowed, reducing the relative gap from EC-Cache.

6.7 Write Performance

Figure 17 shows a comparison of the average write times. The time taken to write an object in EC-Cache involves the time to encode and the time to write out the splits to different workers; Figure 17 depicts the breakdown of the write time in terms of these two components. We observe that EC-Cache is faster than selective replication when writing objects larger than 40 MB, supplementing its faster performance in terms of the read times observed earlier. EC-Cache performs worse for smaller objects due to the overhead of connecting to several machines in parallel. Finally, we observe that the time taken for encoding is less than 10% of the total write time, regardless of the object size.

7 Discussion

While EC-Cache outperforms existing solutions both in terms of latency and load balancing, our current implementation has several known limitations. We believe that addressing these limitations will further improve EC-Cache's performance.

7.1 Networking Overheads

A key reason behind EC-Cache being less effective for smaller objects is its communication overhead. More specifically, creating many TCP connections accounts for a constant, non-negligible portion (few milliseconds) of a read's duration. This factor is more pronounced for smaller read requests which generally have shorter durations. Using long-running, reusable connections may allow us to support even smaller objects. Furthermore, multiplexing will also help in decreasing the total number of TCP connections in the cluster.

7.2 Reducing Bandwidth Overhead

EC-Cache has 10% bandwidth overhead in our present setup. While this overhead does not significantly impact performance during non-peak hours, it can have a non-negligible impact during the peak. In order to address this issue, one may additionally employ proactive cancellation [36, 64] that can help reduce bandwidth overheads of speculative reads.

7.3 Time Varying Skew

EC-Cache can handle time-varying popularity skew and load imbalance by changing the number of parity splits of objects. However, we have not yet implemented this feature due to a limitation posed by Alluxio. In our current implementation, we store individual splits of an object as part of the file abstraction in Alluxio to reduce metadata overheads (§4.2). Since Alluxio does not currently offer support for appending to a file once the file is closed (`ALLUXIO-25` [14]), we cannot dynamically change the number of parities and adapt to time-varying skew. Assuming the presence of underlying support for appending, we expect EC-Cache to respond to time-varying skews better than selective replication. This is because the overhead of any object can be changed in fractional increments in EC-Cache as opposed to the limitation of having only integral increments in selective replication.

7.4 Choice of parameters

Although EC-Cache performs well for a wide range of parameters in our evaluation (§6.6), we outline a few rules of thumb for choosing its parameter values below.

The value of parameter k is chosen based on the size of the object and cluster characteristics: a higher value of k provides better load balancing but negatively affects tail latencies for too large values (as shown in Figure 13 and Figure 14). In general, the larger the size of an object, the higher the value of k it can accommodate without resulting in too small-sized splits and without adversely affecting the tail latency. In our evaluations, we observed $k = 10$ to perform well for a wide range of object sizes (Figure 12).

Suitable choices for Δ depend on the choice of k. As discussed in Section 6.6.2, a higher value of Δ is needed for higher values of k in order to rein in tail latencies. At the same time, each additional read results in a proportional increase in the bandwidth overhead, which would degrade performance for too large a value. In our evaluations, we observed $\Delta = 1$ to be sufficient for $k = 10$ (Figure 10 and Figure 12).

The value of parameter r for each object is chosen based on the skew in object popularity (§6.1).

8 Related Work

A key focus of this work is to demonstrate and validate a new application of erasure coding, specifically in in-memory caching systems, to achieve load balancing and to reduce the median and tail read latencies. The basic building blocks employed in EC-Cache are simple and have been studied and employed in various other systems and settings. We borrow and build on the large body of existing work in this area. However, to the best of our knowledge, EC-Cache is the first object caching system that employs erasure coding to achieve load balancing and to reduce read latencies.

Caching in Data-Intensive Clusters Given that reading data from disks is often the primary bottleneck in data analytics [3, 24, 37, 49, 56, 88, 89, 91], caching frequently used data has received significant attention in recent years [21, 23, 56, 89]. However, existing caching solutions typically keep a single copy of data to increase the memory capacity, which leaves them vulnerable to popularity skew, background load imbalance, and failures, all of which result in disk accesses.

(Selective) Replication Replication is the most common technique for guarding against performance degradation in the face of popularity skew, background load imbalance, and failures [29, 31, 42, 81]. Giving every object an identical replication factor, however, wastes capacity in the presence of skew, and selective replication [20, 63] forms a better option in this case. However, selective replication has a number of drawbacks (§2.3) that EC-Cache overcomes.

Erasure Coding in Storage Systems For decades, disk arrays have employed erasure codes to achieve space-efficient fault tolerance in RAID systems [65]. The benefits of erasure coding over replication to provide fault tolerance in distributed storage systems has also been well studied [85, 93], and erasure codes have been employed in many related settings such as network-attached-storage systems [18], peer-to-peer storage systems [54, 74], etc. Recently, erasure coding has been widely used for storing relatively *cold* data in datacenter-scale distributed storage systems [46, 61, 86] to achieve fault tolerance while minimizing storage requirements. While some of these storage systems [61, 70, 79] encode across objects, others employ self-coding [80, 86]. However, the purpose of erasure coding in these systems is to achieve storage-efficient fault tolerance, while the focus of EC-Cache is on load balancing and reducing the median and tail read latencies. Aggarwal et al. [17] proposed augmenting erasure-coded disk-based storage systems with a cache at the proxy or client side to reduce latency. In contrast, EC-Cache directly applies erasure coding on objects stored in cluster caches to achieve load balancing and to reduce latency when serving objects from memory.

Late binding Many systems have employed the technique of sending additional/redundant requests or running redundant jobs to rein in tail latency in various settings [22, 36, 40, 66, 78, 83]. The effectiveness of late binding for load balancing and scheduling has been well known and well utilized in many systems [60, 64, 82]. Recently, there have also been a body of theoretical work that analyzes the performance of redundant requests [41, 50, 57, 75, 76, 84].

In-Memory Key-Value Stores A large body of work in recent years has focused on building high-performance in-memory key-value (KV) stores [10, 13, 15, 38, 39, 53, 58, 63]. EC-Cache focuses on a different workload where object sizes are much larger than typical values in these KV stores. However, EC-Cache may be used as a caching layer for holding slabs, where each slab contain many key-value pairs. While KV stores have typically employed replication for fault tolerance, a recent work [92] uses erasure coding to build a fault-tolerant in-memory KV store. The role of erasure coding in [92] is to provide space-efficient fault tolerance, whereas EC-Cache employs erasure coding toward load balancing and reducing the median and tail read latencies.

9 Conclusion

Caching solutions used in conjunction with modern object stores and cluster file systems typically rely on uniform or selective replication that do not perform well in the presence of skew in data popularity, imbalance in network load, or failures of machines and software, all of which are common in large clusters. In EC-Cache, we employ erasure coding to overcome the limitations of selective replication and provide significantly better load balancing and I/O performance for workloads with immutable data.

EC-Cache employs self-coding, where each object is divided into k splits and stored in a $(k+r)$ erasure-coded form. The encoding is such that *any* k of the $(k+r)$ splits are sufficient to read an object. Consequently, EC-Cache can leverage the power of choices through late binding: instead of reading from k splits, it reads from $(k+\Delta)$ splits and completes reading an object as soon as the first k splits arrive. The value of Δ can be as low as 1.

The combination of self-coding and late binding, along with fast encoding/decoding using Intel's ISA-L library, allows EC-Cache to significantly outperform the optimal selective replication solution. For instance, for objects of size 40 MB, EC-Cache outperforms selective replication by 3.3× in terms of cache load balancing, and decreases the median and tail read latencies by more than 2×. EC-Cache achieves these improvements while using

the same amount of memory as selective replication. The relative performance of EC-Cache improves even more in the presence of background/network load imbalance and server failures, and for larger objects.

In conclusion, while erasure codes are commonly used in disk-based storage systems to achieve fault tolerance in a space-efficient manner, EC-Cache demonstrates their effectiveness in a new setting (in-memory object caching) and toward new goals (load balancing and improving the median and tail read latencies).

10 Acknowledgments

We thank our shepherd, Andrea Arpaci-Dusseau, and the anonymous reviewers for their valuable feedback. This research is supported in part by DHS Award HSHQDC-16-3-00083, NSF CISE Expeditions Award CCF-1139158, DOE Award SN10040 DE-SC0012463, and DARPA XData Award FA8750-12-2-0331, NSF grant 1409135 and gifts from Amazon Web Services,Google, IBM, SAP, The Thomas and Stacey Siebel Foundation, Apple Inc., Arimo, Blue Goji, Bosch, Cisco, Cray, Cloudera, Ericsson, Facebook, Fujitsu, HP, Huawei, Intel, Microsoft, Mitre, Pivotal, Samsung, Schlumberger, Splunk, State Farm and VMware. Mosharaf and Jack were supported in part by National Science Foundation (grants CNS-1563095 and CCF-1629397) and Google.

References

[1] Amazon EC2. http://aws.amazon.com/ec2.

[2] Amazon Simple Storage Service. http://aws.amazon.com/s3.

[3] Apache Hadoop. http://hadoop.apache.org.

[4] AWS Innovation at Scale. https://www.youtube.com/watch?v=JIQETrFC_SQ.

[5] DFS-Perf. http://pasa-bigdata.nju.edu.cn/dfs-perf.

[6] Evolution of the Netflix Data Pipeline. http://techblog.netflix.com/2016/02/evolution-of-netflix-data-pipeline.html.

[7] Hadoop platform as a service in the cloud. http://goo.gl/11zFs.

[8] Implement the Hitchhiker erasure coding algorithm for Hadoop. https://issues.apache.org/jira/browse/HADOOP-11828.

[9] Intel Storage Acceleration Library (Open Source Version). https://goo.gl/zkVl4N.

[10] MemCached. http://www.memcached.org.

[11] OpenStack Swift. http://swift.openstack.org.

[12] Presto: Distributed SQL Query Engine for Big Data. https://prestodb.io.

[13] Redis. http://www.redis.io.

[14] Support append operation after completing a file. https://alluxio.atlassian.net/browse/ALLUXIO-25.

[15] R. Agarwal, A. Khandelwal, and I. Stoica. Succinct: Enabling queries on compressed data. In *NSDI*, 2015.

[16] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. BlinkDB: Queries with bounded errors and bounded response times on very large data. In *EuroSys*, 2013.

[17] 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 *ICDCS*, 2016.

[18] M. Aguilera, R. Janakiraman, and L. Xu. Using erasure codes efficiently for storage in a distributed system. In *DSN*, 2005.

[19] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, F. Matus, R. Pan, N. Yadav, and G. Varghese. CONGA: Distributed congestion-aware load balancing for datacenters. In *SIGCOMM*, 2014.

[20] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, and E. Harris. Scarlett: Coping with skewed popularity content in mapreduce clusters. In *EuroSys*, 2011.

[21] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Disk-locality in datacenter computing considered irrelevant. In *HotOS*, 2011.

[22] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Why let resources idle? Aggressive cloning of jobs with dolly. In *USENIX HotCloud*, June 2012.

[23] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. PACMan: Coordinated memory caching for parallel jobs. In *NSDI*, 2012.

[24] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in mapreduce clusters using Mantri. In *OSDI*, 2010.

[25] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: Relational data processing in Spark. In *SIGMOD*, 2015.

[26] M. Asteris, D. Papailiopoulous, A. Dimakis, R. Vadali, S. Chen, and D. Borthakur. XORing elephants: Novel erasure codes for big data. In *PVLDB*, 2013.

[27] L. A. Barroso, J. Clidaras, and U. Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture*, 8(3):1–154, 2013.

[28] P. Bodik, I. Menache, M. Chowdhury, P. Mani, D. Maltz, and I. Stoica. Surviving failures in bandwidth-constrained datacenters. In *SIGCOMM*, 2012.

[29] D. Borthakur. The Hadoop distributed file system: Architecture and design. Hadoop Project Website, 2007.

[30] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, et al. Windows Azure Storage: A highly available cloud storage service with strong consistency. In *SOSP*, 2011.

[31] R. Chaiken, B. Jenkins, P. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: Easy and efficient parallel processing of massive datasets. In *VLDB*, 2008.

[32] Y. Chen, S. Alspaugh, and R. Katz. Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads. In *VLDB*, 2012.

[33] M. Chowdhury, S. Kandula, and I. Stoica. Leveraging endpoint flexibility in data-intensive clusters. In *SIGCOMM*, 2013.

[34] M. Chowdhury, Y. Zhong, and I. Stoica. Efficient coflow scheduling with Varys. In *SIGCOMM*, 2014.

[35] A. Cidon, A. Eisenman, M. Alizadeh, and S. Katti. Cliffhanger: Scaling performance cliffs in web memory caches. In *NSDI*, 2016.

[36] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[37] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[38] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. FaRM: Fast remote memory. In *NSDI*, 2014.

[39] B. Fan, D. G. Andersen, and M. Kaminsky. MemC3: Compact and concurrent memcache with dumber caching and smarter hashing. In *NSDI*, 2013.

[40] T. Flach, N. Dukkipati, A. Terzis, B. Raghavan, N. Cardwell, Y. Cheng, A. Jain, S. Hao, E. Katz-Bassett, and R. Govindan. Reducing web latency: The virtue of gentle aggression. In *SIGCOMM*, 2013.

[41] K. Gardner, S. Zbarsky, S. Doroudi, M. Harchol-Balter, and E. Hyytia. Reducing latency via redundant requests: Exact analysis. *ACM SIGMETRICS Performance Evaluation Review*, 43(1):347–360, 2015.

[42] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *SOSP*, 2003.

[43] P. Gopalan, C. Huang, H. Simitci, and S. Yekhanin. On the locality of codeword symbols. *IEEE Transactions on Information Theory*, Nov. 2012.

[44] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: A scalable and flexible data center network. In *SIGCOMM*, 2009.

[45] Y.-J. Hong and M. Thottethodi. Understanding and mitigating the impact of load imbalance in the memory caching tier. In *SoCC*, 2013.

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

[47] Q. Huang, K. Birman, R. van Renesse, W. Lloyd, S. Kumar, and H. C. Li. An analysis of Facebook photo caching. In *SOSP*, 2013.

[48] Q. Huang, H. Gudmundsdottir, Y. Vigfusson, D. A. Freedman, K. Birman, and R. van Renesse. Characterizing load imbalance in real-world networked caches. In *ACM HotNets*, 2014.

[49] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *SOSP*, 2009.

[50] G. Joshi, Y. Liu, and E. Soljanin. On the delay-storage trade-off in content download from

coded distributed storage systems. *IEEE JSAC*, 32(5):989–997, 2014.

[51] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken. The nature of datacenter traffic: Measurements and analysis. In *IMC*, 2009.

[52] S. Kandula, A. Shanbhag, A. Vitorovic, M. Olma, R. Grandl, S. Chaudhuri, and B. Ding. Quickr: Lazily approximating complex adhoc queries in bigdata clusters. In *SIGMOD*, 2016.

[53] A. Khandelwal, R. Agarwal, and I. Stoica. Blowfish: Dynamic storage-performance tradeoff in data stores. In *NSDI*, 2016.

[54] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, et al. Oceanstore: An architecture for global-scale persistent storage. *ACM Sigplan Notices*, 35(11):190–201, 2000.

[55] J. Lee, Y. Turner, M. Lee, L. Popa, S. Banerjee, J.-M. Kang, and P. Sharma. Application-driven bandwidth guarantees in datacenters. In *SIGCOMM*, 2014.

[56] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *SoCC*, 2014.

[57] G. Liang and U. Kozat. Fast cloud: Pushing the envelope on delay performance of cloud storage with coding. *arXiv:1301.1294*, Jan. 2013.

[58] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *NSDI*, 2014.

[59] S. Lin and D. Costello. *Error control coding*. Prentice-hall Englewood Cliffs, 2004.

[60] M. Mitzenmacher, A. W. Richa, and R. Sitaraman. The power of two random choices: A survey of techniques and results. *Handbook of Randomized Computing*, pages 255–312, 2001.

[61] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, and S. Kuamr. f4: Facebook's warm BLOB storage system. In *OSDI*, 2014.

[62] E. Nightingale, J. Elson, O. Hofmann, Y. Suzue, J. Fan, and J. Howell. Flat Datacenter Storage. In *OSDI*, 2012.

[63] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling Memcache at Facebook. In *NSDI*, 2013.

[64] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: Distributed, low latency scheduling. In *SOSP*, 2013.

[65] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *SIGMOD*, 1988.

[66] M. J. Pitkänen and J. Ott. Redundancy and distributed caching in mobile DTNs. In *MobiArch*, 2007.

[67] Q. Pu, H. Li, M. Zaharia, A. Ghodsi, and I. Stoica. Fairride: Near-optimal, fair cache sharing. In *NSDI*, 2016.

[68] K. Rashmi, P. Nakkiran, J. Wang, N. B. Shah, and K. Ramchandran. Having your cake and eating it too: Jointly optimal erasure codes for I/O, storage, and network-bandwidth. In *FAST 15*, 2015.

[69] K. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran. A hitchhiker's guide to fast and efficient data reconstruction in erasure-coded data centers. In *SIGCOMM*, 2015.

[70] K. V. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran. A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the Facebook warehouse cluster. In *USENIX HotStorage*, 2013.

[71] K. V. Rashmi, N. B. Shah, and P. V. Kumar. Optimal exact-regenerating codes for the MSR and MBR points via a product-matrix construction. *IEEE Transactions on Information Theory*, 57(8):5227–5239, Aug. 2011.

[72] K. V. Rashmi, N. B. Shah, and K. Ramchandran. A piggybacking design framework for read-and download-efficient distributed storage codes. In *IEEE International Symposium on Information Theory*, 2013.

[73] I. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[74] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: The OceanStore prototype. In *FAST*, 2003.

[75] N. B. Shah, K. Lee, and K. Ramchandran. The MDS queue: Analysing the latency performance of erasure codes. In *IEEE International Symposium on Information Theory*, 2014.

[76] N. B. Shah, K. Lee, and K. Ramchandran. When do redundant requests reduce latency? *IEEE Transactions on Communications*, 64(2):715–722, 2016.

[77] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, J. Provost, J. Simmons, E. Tanda, J. Wanderer, U. Hölzle, S. Stuart, and A. Vahdat. Jupiter rising: A decade of Clos topologies and centralized control in Google's datacenter network. SIGCOMM, 2015.

[78] C. Stewart, A. Chakrabarti, and R. Griffith. Zoolander: Efficiently meeting very strict, low-latency SLOs. In *USENIX ICAC*, 2013.

[79] A. Thusoo, Z. Shao, S. Anthony, D. Borthakur, N. Jain, J. S. Sarma, R. Murthy, and H. Liu. Data warehousing and analytics infrastructure at Facebook. In *SIGMOD*, 2010.

[80] C. Ungureanu, B. Atkin, A. Aranya, S. Gokhale, S. Rago, G. Calkowski, C. Dubnicki, and A. Bohra. HydraFS: A high-throughput file system for the HYDRAstor content-addressable storage system. In *FAST*, 2010.

[81] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *OSDI*, 2004.

[82] S. Venkataraman, A. Panda, G. Ananthanarayanan, M. J. Franklin, and I. Stoica. The power of choice in data-aware cluster scheduling. In *OSDI*, 2014.

[83] A. Vulimiri, O. Michel, P. Godfrey, and S. Shenker. More is less: Reducing latency via redundancy. In *ACM HotNets*, 2012.

[84] D. Wang, G. Joshi, and G. Wornell. Efficient task replication for fast response times in parallel computation. In *SIGMETRICS*, 2014.

[85] H. Weatherspoon and J. D. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *IPTPS*, 2002.

[86] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *OSDI*, 2006.

[87] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, 2008.

[88] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys*, 2010.

[89] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.

[90] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant stream computation at scale. In *SOSP*, 2013.

[91] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving MapReduce performance in heterogeneous environments. In *OSDI*, 2008.

[92] H. Zhang, M. Dong, and H. Chen. Efficient and available in-memory KV-store with hybrid erasure coding and replication. In *FAST*, 2016.

[93] Z. Zhang, A. Deshpande, X. Ma, E. Thereska, and D. Narayanan. Does erasure coding have a role to play in my data center? Technical Report Microsoft Research MSR-TR-2010, 2010.