2018 IEEE 38th International Conference on Distributed Computing Systems

EC-Store: Bridging the Gap Between Storage and Latency in Distributed Erasure Coded Systems

Michael Abebe, Khuzaima Daudjee, Brad Glasbergen, Yuanfeng Tian
Cheriton School of Computer Science, University of Waterloo
{mtabebe, kdaudjee, bjglasbe, y48tian}@uwaterloo.ca

***Abstract*—Cloud storage systems typically choose between replicating or erasure encoding data to provide fault tolerance. Replication ensures that data can be accessed from a single site but incurs a much higher storage overhead, which is a costly downside for large-scale storage systems. Erasure coding has a lower storage requirement but relies on encoding/decoding and distributed data retrieval, which can result in straggling requests that increase response times. We propose strategies for data access and data movement within erasure-coded storage systems that significantly reduce data retrieval times. We present EC-Store, a system that incorporates these dynamic strategies for data access and movement based on workload access patterns. Through detailed evaluation using two benchmark workloads, we show that EC-Store incurs significantly less storage overhead than replication while achieving better performance than both replicated and erasure-coded storage systems.**

***Keywords*-distributed storage, erasure coding, replication, data movement, data placement**

I. Introduction

The need to store and retrieve big data has fueled the development and adoption of cloud storage systems. In cloud deployments, however, machines frequently experience downtime. For example, Google observed that at any point in time, up to 5% of the nodes within their storage system were unavailable [12]. To ensure data remains available in the presence of these failures, systems must be fault tolerant. Large-scale distributed storage systems typically provide fault tolerance either by replicating [4,14] or erasure encoding data [11,15,19,23,30,52]. Replication creates complete copies of data, incurring a significant storage overhead over erasure coding that partitions data and stores the partitions and their parity fragments on multiple nodes to provide the same level of fault tolerance as replication. Consequently, while erasure encoding stores less data, accessing it requires multi-node retrieval resulting in an increase in data access cost compared to replication [51].

To demonstrate that performance in erasure-coded distributed storage systems is largely determined by the cost of data retrieval, we show a breakdown of average response times in Figure 1 for a workload that retrieves multiple 100 KB blocks.[1] The response time is divided into four categories: the cost of locating data (metadata access), determining which data chunks to retrieve (access planning), retrieving data, and decoding data. As Figure 1 shows, the performance difference between replication and erasure coding is primarily due to

[1]Details are in Section VI.

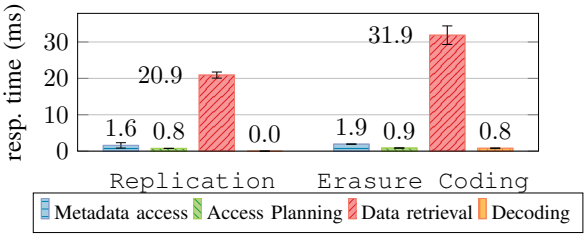

Fig. 1: Response time breakdown for replication and erasure coding under skewed access. Data retrieval times dominate the overall response time.

the time it takes to retrieve data, which dominates overall response times. However, while both systems can tolerate the same number of faults (two, in the example of Figure 1), the replicated system stores 50% more data than the erasure-coded system. These differences motivate a fault tolerant storage system that can achieve the best of both worlds: low storage overhead *and* low latency data retrieval.

When compared to replicated data stores, retrieval costs are higher for erasure-coded storage systems because of the effects of stragglers: the time taken to retrieve the slowest, or *straggling*, data chunk dominates retrieval time [19,29]. Even when parallelism is leveraged, straggler effects are more pronounced in systems that must wait for multiple requests to complete (e.g. in erasure-coded storage) than in systems that wait for only a single request to complete (e.g. in replicated storage) [9,26,46,49,53]. Given that large-scale storage systems are typically deployed in distributed environments, concurrent clients issuing requests in parallel over the distributed storage system inevitably result in the occurrence of stragglers [9].

In our erasure-coded storage system, **EC-Store**, we propose a novel approach to the stragglers problem: intelligently select chunks to retrieve so as to avoid stragglers. This *dynamic data access strategy* uses chunk location information to generate a cost-effective strategy on-the-fly for data retrieval. By incorporating this strategy in EC-Store, we reduce data access latencies and satisfy our best of both worlds goal.

To mitigate the effects of stragglers, some systems use a late binding strategy [19,38,49] in which additional requests are made and the slowest responses are ignored. Late binding can reduce response time but places additional load on the system: responses that will be ignored must still be generated. In contrast, EC-Store's dynamic data access strategy offers excellent performance and places little additional load on the

2575-8411/18/$31.00 ©2018 IEEE
DOI 10.1109/ICDCS.2018.00034

system. Moreover, if additional load can be tolerated, EC-Store can incorporate late binding by intelligently issuing additional requests and ignoring the slowest responses.

EC-Store is designed to take advantage of workloads that contain multi-item retrievals. Such workloads are common [21, 31,39]; for example, all images on a web page are retrieved from a storage system. The items in multi-item requests are often correlated by application semantics [21,28,39]. EC-Store leverages these correlated access patterns to dynamically co-locate data items that are accessed together, thereby reducing chunk retrieval times. EC-Store considers the location of the chunks and their correlations to generate effective data placement and flexible data *access plans* that enable efficient data retrieval.

The contribution of our work is three-fold: *(i)* We demonstrate how the performance of erasure-coded storage systems can be improved significantly through dynamic data placement and data access to mitigate retrieval costs (Section III). We formulate our data access and movement strategies as cost functions with the aim of minimizing expected cost (Section IV). *(ii)* We present EC-Store, a system that incorporates the efficient design and implementation of our techniques (Section V). *(iii)* Through detailed evaluation of EC-Store using both the Yahoo! Cloud Serving Benchmark [7] and a real workload trace of Wikipedia image accesses [47], we validate that our techniques deliver low overhead, fault tolerant, low latency data access for storage systems (Section VI).

II. Erasure Coded Storage and Fault Tolerance

Formally, systems provide r-fault tolerance if they can tolerate r independent faults.[2] Replicated storage systems provide r-fault tolerance under the fail silent assumption [5] by writing data to $r+1$ different locations. Therefore, if r copies of the data become unavailable due to failures, one copy remains available. Many systems default to storing a total of three copies of each data item [4,14].

Erasure codes can provide the **same or better** fault tolerance guarantees than replication, but with significantly **lower** storage requirements [51]. Maximum distance separable codes, such as Reed-Solomon (RS) codes, create $k+r$ chunks of data from an original block of data so that it is possible to recreate the original data from any of the k chunks [40]. Data encoded with these codes are able to tolerate the unavailability of any r of its $k+r$ chunks of data, and is therefore r-fault tolerant.

For a block of data encoded with RS codes parameterized by k and r (denoted as RS(k,r)), the original block of data is divided into $k \geq 2$ chunks from which r parity chunks are generated. Therefore, storing data that has been encoded requires a factor of $\frac{k+r}{k}$ times the amount of storage needed for a single copy of the data. Because replication stores $r+1$ times the amount of data, and $k \geq 2$, erasure coding uses less space to provide the same level of fault tolerance. However, unlike replication, erasure coding requires that a block access retrieves k of the blocks' chunks in parallel followed by a decoding step to reconstruct the data. In this paper, we focus on encoding schemes that access k chunks without the presence of a complete copy of a data block, such as in Facebook's f4 [30] and Windows Azure Storage [19], because these schemes reduce the storage overhead.

[2]We avoid using the terminology k-fault tolerance, as the variables k and r conflict with standard notation used to describe erasure coding.

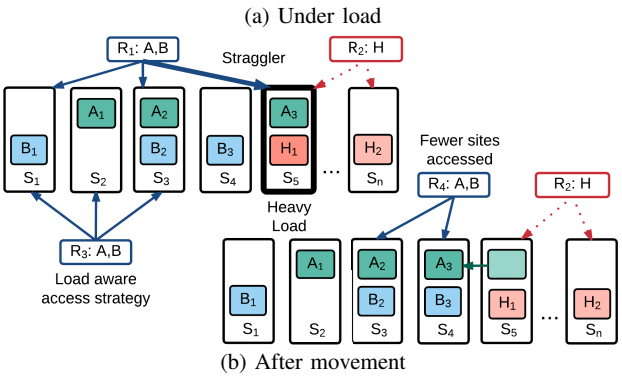

Fig. 2: Data access and movement strategies improve performance. Client R_3 is offered a load aware strategy and does not access the overloaded site S_5. Client R_4 accesses fewer sites after A_3 is moved from S_5 to site S_4.

Like other erasure coded storage systems, EC-Store provides an interface to access and store data as blocks [4,11, 19,52]. Each block is identified by a primary key and has a well-known size, although EC-Store can flexibly accommodate used-defined block sizes.

III. Motivating Example

Straggling chunks lead to stalls in data retrieval for erasure-coded storage: a client cannot continue until all k chunks have been retrieved. Thus, one slow chunk retrieval request can drastically increase the overall request latency. Straggling chunks occur when a site is unable to keep up with the rate that other sites service retrieval requests. Therefore, where data is placed — and how it is accessed — plays a crucial role in performance. We demonstrate the effect that dynamic data access and movement strategies can have on performance using a simple example. Figure 2 shows the storage of data using an RS(2,1) encoding scheme. This scheme requires storing each data item as three chunks, and reconstructing the data item requires retrieving any two of its chunks.

In Figure 2a, data item A has chunks A_1, A_2, A_3 stored at sites S_2, S_3, S_5 respectively. Suppose a client R_1 wishes to retrieve data items A and B. The arrows indicate that R_1 retrieves data from sites S_1, S_3 and S_5. The presence of a popular chunk H_1 on site S_5, accessed by a second client R_2, causes S_5 to be under higher load. Consequently, R_1's request at S_5 is slow and the request becomes a straggler. An alternative, load-aware data access strategy is offered to client R_3, allowing it to avoid requesting a chunk of A from S_5 and instead retrieve it from site S_2. This data access strategy avoids accessing the overloaded site S_5 and waiting for a straggling chunk. We can improve performance further by moving chunk A_3 from site S_5 to S_4, as in Figure 2b. This movement allows for co-located data access for client R_4 that can now retrieve

A and B by accessing only two sites. By accessing fewer sites, the number of requests that must be issued and serviced is reduced thereby reducing the likelihood of stragglers and consequently improving performance [9]. The movement of A_3 will reduce load on S_5, resulting in additional improvement in performance for client R_2.

Although Figure 2 presents a simple motivating example, reality presents a much more complex and challenging environment. In large-scale storage systems, data is distributed across many sites, and access patterns resulting from concurrent client workloads typically overlap. As a result, the objectives to co-locate data and maintain even load distribution can conflict. Erasure coding provides great flexibility for how data should be accessed and placed. We leverage this flexibility by constructing a **cost model** that captures and compares different data access and placement options to select the most effective strategy for minimizing access cost. In the next section, we describe how we formulate this model and solve the data access and movement problem, allowing us to overcome the aforementioned challenges.

IV. Dynamic Data Access and Movement Strategies

In addition to load imbalance, several other factors can also cause straggling chunks. For instance, the rate at which data is retrieved from a storage device, how quickly data is sent over the network, and the rate at which client requests are serviced all contribute to slow chunk retrieval times. Load imbalance itself can be the result of skew in item popularity, data size, or the number of items that are retrieved [46,53].

With these factors in mind, we have developed data access strategies that minimize expected retrieval time. As shown in Figure 2, our data access strategy reduces the number of distributed requests by accessing sites that have co-located data. By placing data so as to minimize distributed access, we develop a data movement strategy that promotes data co-location based on application access patterns. To ensure that load remains evenly distributed, our data movement strategy uses load statistics to relocate data to spread load away from overloaded sites. Finally, as data access and movement strategies allow for alternate choices, we develop our strategies as optimization problems. This optimization formulation quantifies the effects of different data movement choices and allows us to efficiently select access strategies that minimize expected costs, enabling the selection of data movement plans that improve both data access performance and load balance.

As our formulations for data access and movement strategies rely on descriptions of system state, we introduce notation to describe state next.

A. Notation

Table I summarizes our notation. We refer to an individual block as B_i, and the size of its chunks as z_i. We refer to the j-th site (physical machine) as S_j. We use a binary variable $c_{i,j}$ to indicate whether a chunk of B_i is present at site S_j. Note that if B_i is a (k_i, r_i)-encoded block, from the definition of (k_i, r_i) erasure coding, there are exactly $k_i + r_i$ such $c_{i,j}$ that are 1. That is, two chunks from the same block cannot be located at the same site as otherwise the r-fault tolerance guarantees are violated. We consider the state of the system to be a $numBlocks \times numSites$ matrix C with entries $c_{i,j}$ representing the chunk placements.

Chunk and Site Selection Variables	
$C = c_{i,j} = 1$	State of blocks in system: B_i has a chunk at S_j
$s_{i,j} = 1$	B_i's chunk at S_j was selected to be read
$a_j = 1$	Site S_j was selected to be accessed.
Formula	
$cost(C, Q)$	The estimated read cost of performing request Q when the system is in state C
$\Delta(C, b, s, d)$	The estimated gain of moving B_b's chunk from site S_s to S_d
Cost Model Parameters	
o_j	The performance overhead of accessing site S_j
m_j	The overhead of performing a read on the storage media present at site S_j
z_i	The size of block B_i's chunks

TABLE I: Notation used and their meaning

To denote access decisions, we use the binary variable $s_{i,j} = 1$ to indicate that a chunk from block B_i located at site S_j was selected for access. To indicate that site S_j was accessed, we use the shorthand $a_j = \bigvee_i s_{i,j}$.

B. Estimating Data Access Cost

For evaluating the cost of performing a read request, EC-Store considers two key factors: the overhead of accessing a remote site, and the retrieval cost of each requested chunk at that site.

Modelling Access Costs: Given an access plan that satisfies a request for a set of blocks $Q = \{B_i\}$, we model the access cost $cost(Q)$ in Equation 1 as follows. For each site S_j that is considered accessed, that is $a_j = 1$, there is an associated overhead of accessing that site denoted by o_j that can be determined dynamically and allows adaptation as system load changes. This overhead includes factors such as the network latency of accessing a remote site, and the time between when a request is received and when it is processed at a remote site. This time interval is influenced by site load — a site under high load will experience a longer delay before a request is finished processing. The cost of retrieving a chunk from a site depends on the rate that I/O can be performed on the storage media at the site, represented by m_j, and the amount of data that is retrieved, z_i. Our model incorporates this access cost by multiplying these factors $(m_j \cdot z_i)$.

$$cost(Q) = \sum_j \left((o_j \cdot a_j) + \sum_{B_i \in Q} (s_{i,j} \cdot m_j \cdot z_i) \right) \quad (1)$$

Satisfying Access Constraints: Given our cost model, a client should minimize the cost of access while satisfying the constraints imposed by the system state, the request, and the erasure coding parameters. To determine the optimal access plan, we minimize $cost(Q)$ over all valid combinations of the binary decision variables $s_{i,j}$. Observe that for any instance of the problem, o_j, m_j and z_i are constant and therefore $cost(Q)$ is a linear function in terms of $s_{i,j}$. Thus, if we can also define linear constraints that generate valid combinations of $s_{i,j}$, we can exploit existing integer linear programming (ILP)

tools to compute an optimal solution. We present two linear inequalities in Equations 2 and 3 that provide the necessary constraints for our ILP formulation.

The first constraint (Equation 2) ensures that any access plan retrieves at least k_i chunks for each block B_i in the request, guaranteeing that the block can be reconstructed by the property of maximum distance separable codes. A chunk must both exist at a site ($c_{i,j} = 1$) and be selected for access ($s_{i,j} = 1$) for it to be counted towards B_i's required k_i chunks for retrieval. The second constraint (Equation 3) enforces that a site s_j is considered accessed, that is $a_j = 1$, if any chunk located at that site is accessed.[3]

$$\sum_j (c_{i,j} \cdot s_{i,j}) \geq k_i, \forall B_i \in Q \tag{2}$$

$$\big(|Q| \cdot a_j\big) - \sum_{B_i \in Q} s_{i,j} \geq 0, \forall S_j \tag{3}$$

Selecting an Optimal Access Plan: Our cost model is used to determine the optimal data access plan and the associated cost of that access. We denote this cost as $cost(C, Q)$ as it depends not only on the query Q but also on the state of the system, C. We formally define the ILP problem as follows:

$$cost(C, Q) = \min_C cost(Q) \quad \text{subject to constraints in Equations 2 and 3.} \tag{4}$$

1) Late Binding

As mentioned earlier, late binding [38] is a complementary approach to our techniques. Late binding requests $k+\delta$ chunks where $0 < \delta \leq r$, but waits for only the first k responses. Our cost model can support late binding with a slight modification. By changing the right hand side of Equation 2 to $\geq k_i+\delta$, our cost model incorporates the late binding strategy by enforcing a data access plan that will retrieve $k_i+\delta$ chunks for each block in the request. Furthermore, our cost model will optimize the request for an additional δ chunks so that data access has the lowest cost estimate.

C. Estimating Chunk Movement Gain

As data movement affects data accesses, we use our cost model to estimate access costs after movement. Our data movement strategy considers the effect of moving a single chunk at a time. By doing so, we efficiently compute the expected performance gains from moving the chunk. In this section, we describe our function that quantifies the effect of moving a chunk, deferring discussion to Section IV-D for how this function is used. Similarly to our methodology for estimating the cost of executing a request, by quantifying the effects of movement, we are able to compare multiple data movement strategies and select the best strategy for execution.

EC-Store considers two important factors when measuring how moving a chunk affects the system: how movement affects data access costs, and how it affects the distribution of load to sites that store data. We present each of these factors separately before merging them into a single cost function. Each computation is formulated such that a positive result represents an expected performance improvement, a zero result denotes no expected change in performance, and a negative value implies an expected performance degradation.

Consider the case where we wish to evaluate moving block B_b's chunk from source site S_s to destination site S_d. To ensure r-fault tolerance, site S_d must be chosen so that it does not already contain a chunk of B_b. We represent this change in system state by creating a new state matrix $C^{b,s,d}$ from C.[4]

Change in Data Access Costs: To estimate the change in data access costs, we compare the cost of data access under the existing system state to the cost of the next future state. It is infeasible to consider the effect of chunk movement on all queries, so we assemble a set of queries that reflect how block B_b has been historically accessed. Specifically, we generate queries that are of the form $\{B_b, B_i\}$, such that B_i is a block that has appeared in a prior client query that also contained B_b, that is $\{B_b, B_i\} \subseteq Q$ for some client submitted query Q. To ensure that infrequent accesses are not considered as important as frequent accesses, we weight the change in data access costs by the likelihood that B_b and B_i were accessed together, captured statistically as $\lambda_{b,i} = P(\{B_b, B_i\} \subseteq Q | B_b \in Q)$. Equation 5 summarizes how we quantify the change in data access costs, comparing the before and after costs for each generated query weighted by access likelihood:

$$E(C, b, s, d) = \sum_{b_i \in B} \big(cost(C, \{B_b, B_i\}) - cost(C^{b,s,d}, \{B_b, B_i\})\big) \cdot \lambda_{b,i} \tag{5}$$

For the example in Figure 2, we would observe that accessing data would have lower estimated cost after moving A_3, as in Figure 2b. As a result, this movement plan would have a positive value for $E(C, b, s, d)$.

Quantifying System Load: In addition to the estimated change in data access costs, we consider the effect of moving a chunk on the load of a system. Recall that high load at a site can slow data retrieval requests at that site, which leads to stragglers [19]. To avoid the effects of skewed load, we strive to distributed load evenly across sites. To achieve this load balance, we promote data movement from heavily loaded sites to lightly ones. We model site load before and after data movement to quantify load balance improvements.

We calculate the load at site S_j in state C as $\omega(C, S_j)$ from the site's CPU utilization and I/O load. We observed that these factors were correlated with the rate at which requests were serviced. To model the load at a site after movement, $\omega(C^{b,s,d}, S_j)$, we proportionally shift the CPU utilization and I/O load from the source site to the destination site based on chunk size and chunk access likelihood.

Normalizing System Load: To represent the degree to which a site has diverged from the average load $\tilde{\omega}(C)$, we define a site load balance factor as: $\Omega(C, S_j) = \left|1 - \frac{\omega(C,S_j)}{\tilde{\omega}(C)}\right|$.

[3] If a site is accessed then $\sum_{B_i \in Q} s_{i,j} \geq 1$. However, no more than $|Q|$ chunks can be requested from a single site. Therefore, if the site is accessed then a_j must be set to 1 for the inequality to hold.

[4] To change the system state we set $c^{b,s,d}_{b,s}$ to 0, $c^{b,s,d}_{b,d}$ to 1, and leave other values in $C^{b,s,d}$ unchanged from C.

Algorithm 1 selectMovementPlan

1: blocksUnderConsideration = GETCANDIDATEBLOCKS(); ▷ Recent & frequent blocks
2: $Score_{opt} = 0$; ▷ Init. $B_{b-opt}, S_{s-opt}, S_{d-opt}$ to NULL
3: **for all** Block B_b : blocksUnderConsideration **do**
4: candidateDestinations = GETCANDIDATEDESTINATIONS(B_b); ▷ Exclude sites where B_b's chunks are present
5: **for all** Chunk c : B_b.chunks **do**
6: Site S_s = c.chunk_location; ▷ S_s is the source site;
7: **for all** Site S_d : candidateDestinations **do** ▷ S_d is a potential destination;
8: $Score = \Delta(C, B_b, S_s, S_d)$;
9: **if** $Score > Score_{opt}$ **then**
10: $(Score_{opt}, B_{b-opt}, S_{s-opt}, S_{d-opt}) = (Score, B_b, S_s, S_d)$;
11: **return** $(B_{b-opt}, S_{s-opt}, S_{d-opt})$;

If $\Omega(C, S_j) = 0$ then S_j has exactly average load, but as $\Omega(C, S_j)$ continues to increase, S_j drifts away from average load. In Figure 2a, site S_5 would have a larger $\Omega(C, S_5)$ value than $\Omega(C, S_4)$ as S_5 is under higher than average load.

Estimating Change in System Load: We consider the effect of load created by data movement on both the source and destination site by using the load balance factor of the most imbalanced site. To represent the load of the most imbalanced site, we construct $\Omega(C, S_s, S_d)$ given by Equation 6:

$$\Omega(C, S_s, S_d) = \max(\Omega(C, S_s), \Omega(C, S_d)) \quad (6)$$

The difference between $\Omega(C, S_s, S_d)$ and $\Omega(C^{b,s,d}, S_s, S_d)$ represents the change in load balance factors at the source and destination as a result of chunk movement, which we summarize as $I(C, b, s, d)$ in Equation 7. In Figure 2, we observe that moving data from site S_5 to S_4 will decrease the load factor of S_5, resulting in a positive $I(C, b, s, d)$ value.

$$I(C, b, s, d) = \Omega(C, S_s, S_d) - \Omega(C^{b,s,d}, S_s, S_d) \quad (7)$$

Estimating Benefit from Chunk Movement:

To combine the two factors, namely, effect of data access and effect on load, we sum Equations 5 and 7. To control the relative influence of the two factors, we use weight parameters w_1 and w_2. The total estimated benefit of moving block B_b's chunk from source site S_s to destination S_d when the system is in state C is represented by $\Delta(C, b, s, d)$ in Equation 8. $\Delta(C, b, s, d)$ is positive if the movement of the chunk is expected to be beneficial and negative if it is expected to worsen costs.

$$\Delta(C, b, s, d) = w_1 \cdot E(C, b, s, d) + w_2 \cdot I(C, b, s, d) \quad (8)$$

D. Selecting Chunks for Movement

The previous section described how expected changes in system performance are measured both in terms of data accesses and load distribution when moving chunks. We select chunks for movement and their target destinations through the use of movement plans. A movement plan consists of a block B_b, a site S_s containing a chunk of B_b, and a candidate destination site for that chunk S_d. As it is infeasible to compute expected performance gains exhaustively for every movement plan (B_b, S_s, S_d), we employ a heuristic strategy for generating movement plans. Our heuristic is guided by two principles: recently accessed blocks are likely to be accessed again [21], and sites that are under heavy load contribute to the straggling chunk problem [19].

We summarize how we select a plan for data movement in Algorithm 1. We first retrieve a set of blocks that are candidates for movement because they have been recently accessed (Line 1). We probabilistically generate this set based on access likelihood, which allows us to explore the effect of moving many other different data items. For each candidate block, we consider moving its chunks to new destination locations (Line 4). Finally, for each candidate movement plan, we compute the expected benefit of executing the move (Line 8).

At any point in time we can halt execution of Algorithm 1 and execute the movement plan that has the best score so far. Therefore, we use early stopping conditions when searching for candidate plans and greedy subroutines that return lists of candidates ordered by the best candidate first. For instance, Line 5 iterates over chunks ordered by site load so that chunks located at the most heavily loaded site are evaluated first.

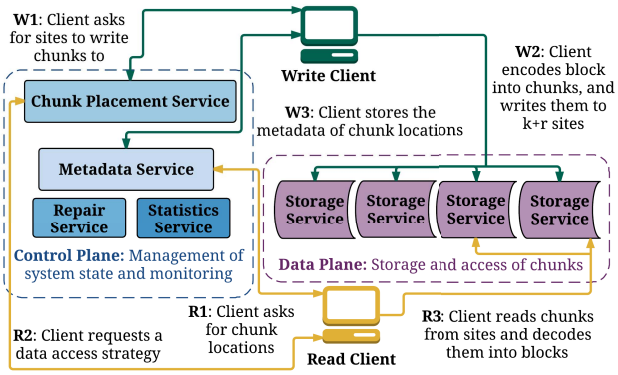

Fig. 3: System architecture showing division of control and data planes including processes for reading (R1-R3) and writing data (W1-W3).

V. EC-STORE: ARCHITECTURE AND IMPLEMENTATION

We have developed EC-Store, a system that incorporates the design and implementation of our data access and movement strategies. Similar to other systems, EC-Store clients store and access data by communicating with a service API to *put* a block, *read* block(s), or *delete* a block [4,11,19,52]. Figure 3 shows the steps involved for reading and writing data. Our system inserts an additional step to decide how data should be placed (W1) and accessed (R2).

EC-Store's architecture is logically separated into two components: a control plane and a data plane. The data plane consists of sites that execute a storage service to manage the storage and retrieval of chunks. The location of each block's encoded chunks within the data plane are managed by the metadata service in the control plane, which implements the strategies described in Section IV.

EC-Store services were developed in C++ and communicate with each other using remote procedure calls through the Apache Thrift library [44]. We use the Jerasure 2.0 library [34] for encoding and decoding blocks. Next, we describe the functionality and implementation of the statistics, chunk placement, and repair services.

A. Statistics Service

To maintain load and access correlation statistics used by the data movement strategies described in Section IV, a statistics service is provided. This service approximates each site's load by tracking the CPU utilization, I/O load, and the number of chunks stored at each site. These values are reported by storage services to the statistics service at regular intervals, and used as ω_j in our data movement strategy. Statistics are reported every 5-10 seconds, which we found provided up-to-date information. The statistics service also maintains information on sampled block access patterns within a sliding interval of previous requests. Each block co-access pair (B_b, B_i) within a sampled request is tracked to compute the conditional likelihood of block co-access, stored as $\lambda_{b,i}$. When the access pattern exits the interval of previous requests, the likelihood of co-access is adjusted so that the statistics service can capture changes in workload. Tracking likelihoods and reporting statistics more frequently would require more space and increase communication overheads but provide higher accuracy. In our experiments, we tracked a sliding interval of 5000 requests which was large enough to capture access correlations.

B. Chunk Placement Service

The chunk placement service consists of two sub-modules that provide EC-Store with strategies for data access and data movement: the chunk read optimizer and the chunk mover.

1) Chunk Read Optimizer

To access data, the client service calls into a local chunk read optimizer, which receives chunk locations for a set of blocks that a client wishes to read and returns the set of chunks that should be retrieved from each site. The read strategy is computed by finding the solution to the $cost(C, Q)$ function as described in Section IV-B using the SCIP library [1] as an ILP solver.

Preliminary experiments showed that solving the ILP problem to decide on a data access strategy took in the order of tens of milliseconds, which is much higher than the access costs that were observed in Figure 1. To address this latency concern, we cache and reuse previous access plans that satisfy a new request for data access. If there is no access plan in the cache that can satisfy the request then we generate an access plan using a greedy heuristic, which works as follows. We decide for each block whether a given chunk will be retrieved based on the state of the existing access plan. If a chunk for the requested block is present at an accessed site, that chunk is added to the access plan. If k chunks for the block have not been added to the access plan, after all previously accessed sites in the plan are examined, sites for the remaining chunks are randomly selected.

An access plan cache miss triggers a background worker thread that solves the data access problem using the ILP solver. Once the ILP solution is computed, it is stored in the cache and replaces the greedy solution, enabling all future requests for the same blocks to use the ILP solution without the need to solve the ILP problem. When the cost parameters in the ILP problem change as a result of new system state, we dynamically reload solutions.

2) Chunk Mover

The chunk mover is responsible for asynchronously moving chunks within the system to improve block co-access performance and load balance. The chunk mover executes Algorithm 1 to select a movement plan to copy the chunk from source to destination. The block metadata is then updated to reflect the new chunk location, and the old chunk can be deleted as all future accesses receive the new chunk location. The chunk mover can throttle the rate at which chunks are moved to ensure that data movement does not add overhead to the system, as we experimentally show in Section VI-C5.

3) Parameter Choices

The data access and movement strategies employed by the chunk placement service are parameterized for generality. The cost model depends on two factors: o_j and m_j that reflect the cost of accessing a site and the cost of reading from a site, respectively. We dynamically set o_j for each site based on the average response time of periodic load-status requests to storage services. These requests are decoupled from data access requests and therefore measure request processing time. Thus, an increase in load proportionally increases the response time of load-status requests. We empirically determined m_j by measuring the time taken to retrieve increasing numbers of chunks from a single site. Because our experiments were conducted on homogeneous hardware, we set m_j to be the same value for each machine. We found that an approximate value of m_j, normalized to an average value of o_j, was $m_j = 1$ when $o_j = 5$. Both the chunk read optimizer and chunk mover receive the o_j and m_j parameters from the statistics service to support dynamic strategies for data access and movement.

In our data movement strategy, w_1 and w_2 weight the expected improvement in query performance and expected improvement in load balance, respectively. The largest value for $I(C, b, s, d)$ is 1, and occurs when the system state changes from unbalanced to perfectly balanced. Figure 2 shows an example of accessing one less site after data movement, which results in $E(C, b, s, d) = avg(o_j)$. Initially, we set $w_1 = 1$ and $w_2 = avg(o_j) = 5$ to balance the factors, but then performed a parameter search by varying w_2. Empirically, we found that $(w_1 = 1, w_2 = 3)$ yielded the best performance, which indicates that expected query performance is the more dominant factor in the data movement strategy.

As discussed in Section II, erasure coded storage systems must choose parameters k and r. By default, we use a $k =$

$2, r = 2$ encoding scheme. Choosing a value of k poses a trade-off: larger values of k reduce the storage overhead, but must access more sites in parallel and therefore incur higher access costs. Hence, $k = 2$ is a popular option for applications aiming to reduce access costs [38].

C. Repair Service

To ensure availability in the presence of node failures, EC-Store reconstructs chunks on sites that have failed. The repair service polls each site's storage service and marks the site unavailable for access if it does not respond. The repair service waits 15 minutes, as in GFS [12], before reconstructing chunks using our data movement strategy to select chunk destinations.

VI. Experimental Evaluation

In this section, we present an experimental evaluation of our EC-Store system. We first describe the experimental setup and then present system performance results.

A. Experimental Setup

The objective of our experiments is to evaluate how our data placement and access strategies improve performance, and to verify that our system operates with low overhead. We compare our techniques against replication (`R`) and traditional erasure coding (`EC`), which use the strategies of random data placement and access [38]. Although strategies exist for reducing replication response times as discussed in Section VII, we use a randomized strategy because it offers a baseline comparable to erasure coding, and it is common in many systems [10,24]. We compare these baselines against our two configurations of data access: (i) erasure coding with our cost model (`EC+C`), and (ii) erasure coding with our cost model and chunk movement (`EC+C+M`). This setup allows us to differentiate the performance contribution of the cost model for data access from that of data movement. To investigate the effects of additional requests, we also evaluate erasure coding with late binding (`EC+LB`) and our data movement and access strategy with late binding (`EC+C+M+LB`). All configurations are implemented into EC-Store.

Our experimental testbed consists of 36 identical machines, of which 32 machines are used to co-locate the I/O intensive storage service with the compute intensive EC-Store client service. The remaining four machines are allocated, one each, to the metadata service, the chunk placement service, the statistics service, and for running the benchmark workload. All machines are within the same local area network, interconnected by a 10 Gb Ethernet link. Each machine runs Ubuntu 16.04, and has 12 physical cores (two Intel E5-2620v2 processors), 32 GB of memory, and 1 TB of local disk storage (Seagate Constellation ES.3 6 Gb/s SATA hard drives).

To tolerate two faults, we use a RS(2,2) block encoding scheme for erasure coding, and keep 3 copies of all blocks for replication. Under these configurations replication stores three times, and erasure coding two times, the amount of original data. Therefore, replication stores 50% more data than erasure coding while providing the same level of fault tolerance.

B. Benchmarks

We use two benchmarks that store and retrieve objects of various sizes to evaluate our data access strategies and techniques. The first benchmark is the popular YCSB-E workload [7] that retrieves ranges of keys together as in a messaging system. Items within the same range are correlated and accessed together, representing a chain of messages within a conversation [7]. The second benchmark is a trace of pages of Wikipedia image accesses [47]. All images on a page are retrieved when a page is loaded, resulting in correlations between images that appear on the same page. The likelihood of retrieving a page of images is derived from the trace and follows a Zipf distribution [47]. Both the number of images on a page and image sizes follow a power law distribution. The median page consisted of about 10 images, and the median image was approximately 500 KB in size. Images on Wikipedia are treated as static resources, making them good candidates for storing within a block storage system.

Both benchmarks employ a configurable number of concurrent clients that submit requests independently with zero think time between requests. Unless otherwise stated, our default number of clients was 100. Our experimental methodology is as follows. For the YCSB-E benchmark workload, the system is first loaded with 1 million blocks of fixed size. A warm up scan workload is run for 20 minutes, accessing keys with a uniform distribution. A second scan workload is then run for 20 minutes over which we collect performance measurements, this time accessing keys using a power law distribution (with default exponent 1) to effect workload change. For our Wikipedia experiments, each trace requests a set of blocks corresponding to the images found on a specific page over a measurement interval of 20 minutes after a warm-up run of the same type and duration as in the YCSB experiments.

C. Experimental Results

In this section, we present and discuss results for our experiments with the YCSB and Wikipedia workloads. All results presented are the average of five runs with bars around the mean representing 95% confidence intervals.

1) Response Time

We study data access performance using a YCSB-E workload and show the average response time, over time in Figure 4a, for each of the configurations. In this figure, response time for our data access strategy (`EC+C`) and our access and movement strategy (`EC+C+M`) start similarly before `EC+C+M` decreases over the first 8 minutes. The movement strategy quickly learns the workload pattern and moves data to balance load and co-locate co-accessed data items that together improve response times. Recall that the workload change results from inducing skew in data item access. Therefore, our movement strategy reduces response times is a consequence of dissipating the effects of skew. By the end of the experiment, we observe that our data access and movement strategies achieve a 40% improvement in latency over baseline erasure coding. Furthermore, we reduce latencies by 20% compared

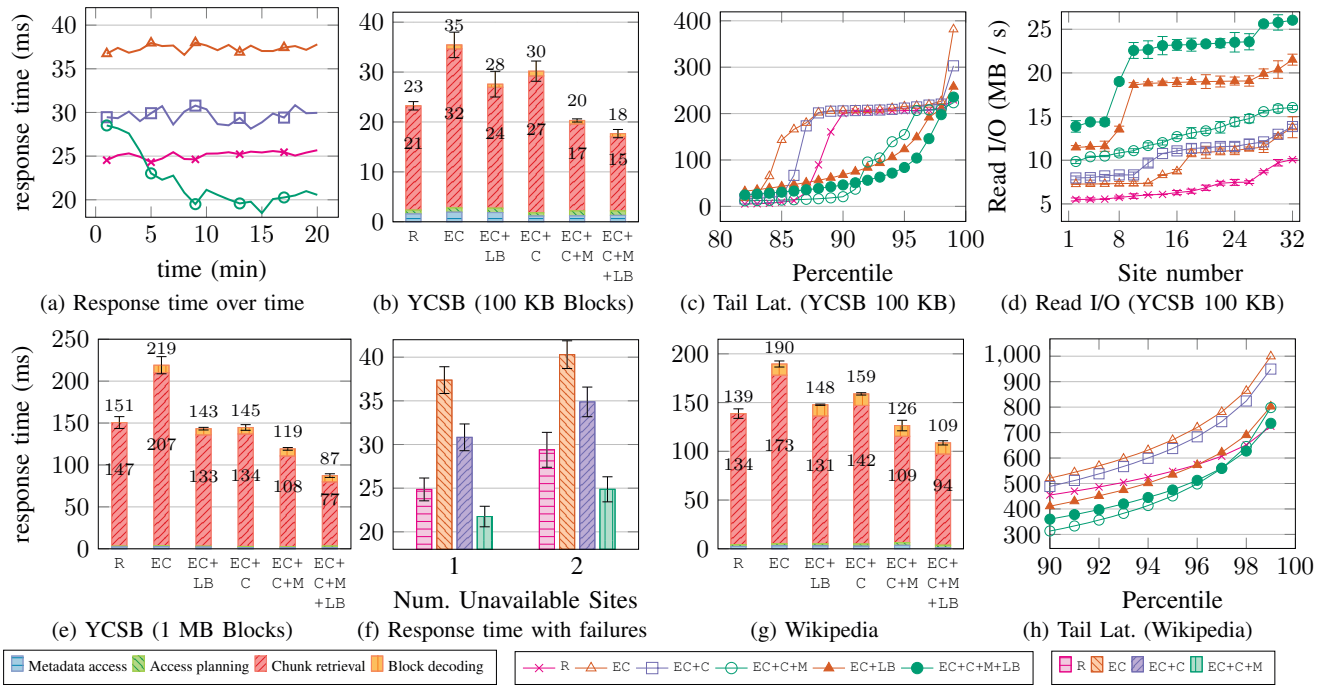

Fig. 4: Experimental results for YCSB-E and Wikipedia experiments. Data access and movement strategies significantly improve performance over standard techniques, by reducing the time taken to retrieve chunks.

to replication while using only two-thirds of the storage space that replication uses. Our data access strategy (EC+C) alone reduces response time by 15% compared to erasure coding, demonstrating the benefit of our cost model.

Figure 4b shows a breakdown of average response time from the YCSB experiment into four categories: accessing metadata, determining an access strategy, retrieving chunks, and decoding the chunks into a block. As noted in Section I, the most dominant factor in the response time is retrieval of chunks from the storage services and not the overhead from block decoding. Metadata access and generation of strategy for data access also make up a small portion of the remaining response time. The figure demonstrates that the improvements come from reducing data retrieval time resulting from the utilization of our techniques.

The figure also includes late binding ($\delta = 1$), a common approach for improving erasure-coded storage data retrieval performance. EC+C+M's proactive data movement and minimization of access costs reduces the time taken to retrieve data by 30% when compared to late binding (EC+LB). Our technique places little extra load on the storage system (Section VI-C5), but late binding increases load by making additional chunk requests (Section VI-C2). Because our techniques and late binding are complimentary, they can be combined as described in Section IV-B1. This tandem approach (EC+C+M+LB) reduces the retrieval latency of late binding (EC+LB) by 40%.

In Figure 4c, we plot a cumulative distribution function (CDF) of the tail response times from the YCSB experiment. We observed that in the lower percentiles, the performance

Technique	R	EC	EC +LB	EC +C	EC +C+M	EC+C +M+LB
λ	45.4	43.0	22.8	31.1	24.5	19.8

TABLE II: λ values for experimental techniques in YCSB 100 KB block experiment. Lower λ values indicate lower levels of load imbalance.

difference between replication and erasure coding is the time taken to decode blocks. At the tail, overall response times increase because chunk retrieval time increases but the other components of response time remain constant. More concretely tail retrieval times are where the effects of straggling chunks are observed, presenting as a sharp increase in latency. Therefore, by examining tail latencies, we can see the effect that different techniques have on reducing stragglers.

As Figure 4c shows, intelligently selecting chunk accesses (EC+C) reduces the percentage of requests that experience the straggler effect compared to baseline erasure coding (EC). Data movement in addition to data access strategies (EC+C+M) mitigate the effects of stragglers further. Data movement is effective because it reduces the number of sites needed for access, and helps balance load. Although late binding (EC+LB) also reduces the effects of stragglers, EC+C+M has a lower 99^{th} percentile latency than EC+LB because the extra requests made by late binding place additional load on the storage sites. We further examine the effects of load in the next section.

2) *Effect on Load*

Figure 4d shows the average amount of data that is read per second from each site during the experiment. While a decrease in latency increases throughput, it causes more data to

be read, which does not always improve performance. For late binding (EC+LB), the additional requests result in more data being read than with our data access and movement techniques (EC+C+M) while EC+C+M has lower data retrieval times. Late binding's increase in the amount of data read is the result of making δ extra chunk requests when retrieving a block, not from performing more block reads than EC+C+M.

Our techniques help reduce the load imbalance experienced in erasure-coded systems. To demonstrate this effect, we define I/O load to be the amount of data read per site, and compare the I/O load imbalance factor λ for each configuration in Table II.[5] As shown, our data access strategy (EC+C) reduces load imbalance compared to both baseline erasure coding and replication. This improvement is due to our cost model that considers site load through the o_j parameter. Intuitively, our cost model creates a feedback loop: lightly loaded sites are preferred for access over heavily loaded sites, so load shifts from heavily loaded sites to lightly loaded sites to reach a steady state of balanced load. Actively moving data (EC+C+M) and considering load when doing so further improves load balance. Late binding can improve load balance [38], and our techniques provide similar levels of balance. However, late binding makes extra requests that increase system load while our techniques place minimal additional load on the system (Section VI-C5), achieving load balance through load-aware data movement and access strategies.

3) Block Size

We also experimented with varying block sizes, both smaller (10 KB) and larger (1 MB), and have observed similar trends in performance as previously described. We show the breakdown of response times for our experiment with 1 MB blocks in Figure 4e. In this experiment, we found that EC+C+M techniques reduced data retrieval latency by a greater margin (nearly 50% over EC, 27% over R, and 21% over EC+LB) than with the smaller block size of 100 KB. The reduction is greater because large blocks are more expensive to retrieve and therefore magnify the effects of load imbalance, and stragglers, that our techniques mitigate. The time taken to generate an access strategy for our techniques (EC+C and EC+C+M) is the same as for replication and erasure coding irrespective of block size. This indicates that our mechanisms for fast access plan generation using the plan cache, which had a 90% hit rate, and a greedy solution on cache miss, are effective in minimizing the latency overhead of the ILP solver. These results confirm that our techniques are effective over a range of block sizes.

4) Fault Tolerance

To show that our techniques are robust in the presence of failures, we performed our 100 KB YCSB experiments when some nodes were unavailable. To do so, we purposefully failed the storage service on n nodes chosen at random, but did not trigger reconstruction of unavailable data. Consequently, requests to unavailable nodes fail and when the failure is detected, requests are routed to only the available nodes. In Figure 4f, we present the average response times for requests after 1 or 2 nodes have failed. Compared to when there are no failures (Figures 4a and 4b), the average response time increases by about 1 ms and 5 ms for all systems when there are 1 and 2 node failures, respectively. Unlike the 1 node failure, when 2 nodes fail some blocks will have chunks located on both of the failed nodes, causing our data access strategy (EC+C) to formulate a data access plan in which the remaining available 2 chunks must be retrieved. However, as our strategy considers the cost of accessing a site, the access plans for the remaining blocks can adapt as site load increases. As our data movement strategy (EC+C+M) moves chunks to help balance overall system load, the relative performance improvements persist when failures occur.

[5] $\lambda = (\frac{L_{max} - L_{avg*}}{L_{avg*}}) * 100$, where the load on the maximally loaded site is denoted by L_{max} and the average load by L_{avg*} [38].

Resource Usage	Statistics	Chunk Read Optimizer	Chunk Mover
Memory	2.8 GB	10.5 MB	80 MB
CPU	<0.5%	0.5%	15%
Network	20 KB/s	<1 KB/s	500 KB/s

TABLE III: Physical resources used by EC-Store.

5) Resource Consumption

Table III summarizes the physical resources used by the chunk placement service (chunk read optimizer and chunk mover) and the statistics service during a YCSB experiment with 1 million 1-megabyte blocks. The chunk placement service uses resources to generate background access plans, manage the plan cache, and to create and cost movement plans. For these components, the resource usage scales with the number of blocks accessed in a request, which is generally small, (e.g. 10 [21,31,39]) and the number of blocks considered for movement, which our movement heuristics limit.

The statistics service uses memory exclusively to track access patterns. As more blocks are stored, the relative amount of space needed for the statistics service decreases because there is a long tail of blocks that are infrequently accessed, that therefore require little space for tracking access correlations. However, if there is a uniform distribution of block accesses then the space required for tracking access correlations can increase substantially. Overall, to provide dynamic data access and movement strategies, EC-Store used only an additional 0.3% of the space needed to store data.

The network overhead of our techniques is also minimal. We limit the chunk mover to moving less than one chunk per second, so the data transfer necessary for dynamic data movement is less than 1 additional block request every second. Consequently, EC-Store incurs a network overhead of less than 0.1% compared to the total data transferred during the benchmark. Comparatively, late binding adds an additional chunk request to every data item access (50% more chunk requests in our experiments). These results show our techniques use meager resources to achieve large performance gains.

6) Wikipedia Results

We performed experiments using the Wikipedia image accesses as described in Section VI-B, and show average response times in Figure 4g. The figure shows that our dynamic

data movement and access strategies (EC+C+M) significantly reduce the time taken to retrieve data: over erasure coding (EC) by 40%, replication (R) by 20%, and late binding (EC+LB) by 17%. Our techniques achieve these improvements without making extra requests that late binding generates, storing 50% less data than replication, and using little additional resources (Table III). The results show that our intelligent data access strategy (EC+C) provides about half of the overall reduction in latency. As with the YCSB experiments, when late binding is combined with our approach (EC+C+M+LB), data retrieval times are reduced by an additional 15%.

Figure 4h shows a tail latency (CDF) graph for the Wikipedia experiments. Unlike the YCSB experiment (Figure 4c), the Wikipedia graph is smoother in appearance and lacks the sharp increase in latency that occurs from stragglers. This difference is due to the distribution in block sizes that appear in the Wikipedia workload but not the YCSB workload. As Section VI-C3 noted, the time taken to retrieve data increases as block size increases. Therefore, a straggling request for small chunks can have the same retrieval latency as a straggler-free request for large chunks. This property makes the effects of stragglers less prominent in a single CDF. We observe that EC+C+M and EC+C+M+LB have the lowest latencies for the entire distribution of requests, and it is only at the extreme tail that late binding (EC+LB) has comparable latency to EC+C+M. These improvements are possible due to our techniques providing a size-aware access and movement strategy to ensure that load remains balanced in the presence of varying I/O costs for different requests.

VII. Related Work

Erasure-coded storage is popularly used by industrial companies that have designed large-scale (distributed) storage systems [11,15,30,32]. Some systems [43,56,57] have used erasure coding to build highly available key-value stores. EC-Store's distributed architecture is motivated by these systems, but differs in its dynamic data access and placement strategies.

To avoid the straggling chunks problem, several systems use a late binding strategy [19,38,49]. Our work demonstrates that dynamic data movement as well as dynamic access strategies, in addition to being complementary to late binding, can provide performance gains over late binding.

To reduce the overhead of encoding, researchers have designed new erasure codes [19,22,36,37,42,48] or exploited their algebraic properties [29]. These approaches do not address strategies for placement and access of encoded data. Theoretical work has examined bounds on latency in erasure-coded storage systems [2,6,41,55]. Instead of a single static placement policy, our work contributes algorithms for dynamic data movement and placement including a practical system using these strategies. This design allows EC-Store to adapt to changing workloads and access patterns.

Data access and placement strategies have been proposed for replicated data [8,17,20,33,35,39,46] but they cannot be used for erasure coded systems as they rely on the assumption that a complete copy of every data item is accessible at a single storage site, which does not hold for encoded data. EC-Store builds upon the ideas of these systems by tracking access history, co-locating data, and dynamically moving data. EC-Store offers erasure-coded storage systems a holistic approach for data access, placement, and movement by incorporating system load, block access patterns, and access frequency into its strategies. In contrast, WPS [50] makes data placement decisions using only chunk access frequency statistics, and does not specify strategies for data access.

The Triones [45] tool proposes to place encoded chunks across data centers to provide availability in the presence of entire data center failures. Triones considers the monetary cost of data access and storage as the primary objective for static data placement while EC-Store dynamically optimizes for data access latencies within storage nodes in a data center.

There has been work that focuses on repair performance by clustering anti-correlated blocks [18,58]. In these approaches, an original copy of data is kept at one site and erasure-coded chunks at other sites for fault tolerance. These encoded chunks are accessed for repair only if the original data copy is unavailable, unlike in EC-Store where encoded data is always accessed. These systems offer access strategies but do not perform dynamic data movement like EC-Store.

Encoding-aware replication [27] supports both encoded and replicated data to decrease chunk access times. Other systems [3,13,16] improve response times by caching chunks or complete copies of data. These approaches increase storage overheads and do not lower response times more than replication, unless the copies are stored on faster storage media. EC-Store outperforms replication without increasing the storage overhead of erasure coding. Other work [25,54] focuses on choosing erasure coding parameters that minimize response times for hot data items at the cost of increased storage overheads by decreasing or increasing k. As described in Section V-B3, EC-Store targets performance improvements of erasure coded storage regardless of choices for k and r.

VIII. Conclusion

We presented EC-Store, a distributed erasure-coded storage system that is designed for dynamic data access and movement. Our optimization-driven approach allows for data accesses that minimize response times. By considering access patterns, EC-Store improves the distribution of load within the system and supports efficient retrieval by co-locating frequently accessed data items. EC-Store reduces the average time to retrieve data by nearly 50% when compared to standard erasure coding techniques, and 30% compared to replication. EC-Store improves the state-of-the-art in storage systems by incorporating the best of both erasure-coded storage and replication to provide fault tolerance, low latency data access *and* low overhead storage.

Acknowledgments

Funding for this project was provided by the Natural Sciences and Engineering Research Council of Canada, the University of Waterloo, Canada Foundation for Innovation, Ontario Research Fund, and the province of Ontario.

REFERENCES

[1] ACHTERBERG, T. Scip: solving constraint integer programs. *Mathematical Programming Computation 1*, 1 (2009), 1–41.

[2] AGGARWAL, V., AL-ABBASI, A. O., FAN, J., AND LAN, T. Taming tail latency for erasure-coded, distributed storage systems. *arXiv preprint arXiv:1703.08337* (2017).

[3] AGGARWAL, V., CHEN, Y.-F. R., LAN, T., AND XIANG, Y. Sprout: A functional caching approach to minimize service latency in erasure-coded storage. In *Distributed Computing Systems (ICDCS), 2016 IEEE 36th International Conference on* (2016), IEEE, pp. 753–754.

[4] BORTHAKUR, D. Hdfs architecture guide, 2008.

[5] BRASILEIRO, F. V., EZHILCHELVAN, P. D., SHRIVASTAVA, S. K., SPEIRS, N. A., AND TAO, S. Implementing fail-silent nodes for distributed systems. *IEEE Transactions on Computers 45*, 11 (1996), 1226–1238.

[6] CHEN, S., SUN, Y., KOZAT, U. C., HUANG, L., SINHA, P., LIANG, G., LIU, X., AND SHROFF, N. B. When queueing meets coding: Optimal-latency data retrieving scheme in storage clouds. In *INFOCOM, 2014 Proceedings IEEE* (2014), IEEE, pp. 1042–1050.

[7] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing* (2010), ACM, pp. 143–154.

[8] CURINO, C., JONES, E., ZHANG, Y., AND MADDEN, S. Schism: a workload-driven approach to database replication and partitioning. *Proceedings of the VLDB Endowment 3*, 1-2 (2010), 48–57.

[9] DEAN, J., AND BARROSO, L. A. The tail at scale. *Communications of the ACM 56*, 2 (2013), 74–80.

[10] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS operating systems review* (2007), vol. 41, ACM, pp. 205–220.

[11] FIKES, A. Storage architecture and challenges. http://static.googleusercontent.com/external_content/untrusted_dlcp/research.google.reverse-proxy.org/en/us/university/relations/facultysummit2010/storage_architecture_and_challenges.pdf, 2010. Accessed: 2010-07-29.

[12] FORD, D., LABELLE, F., POPOVICI, F. I., STOKELY, M., TRUONG, V.-A., BARROSO, L., GRIMES, C., AND QUINLAN, S. Availability in globally distributed storage systems. In *OSDI* (2010), vol. 10, pp. 1–7.

[13] FRIEDMAN, R., KANTOR, Y., AND KANTOR, A. Replicated erasure codes for storage and repair-traffic efficiency. In *Peer-to-Peer Computing (P2P), 14-th IEEE International Conference on* (2014), IEEE, pp. 1–10.

[14] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The google file system. In *ACM SIGOPS operating systems review* (2003), vol. 37, ACM, pp. 29–43.

[15] HAEBERLEN, A., MISLOVE, A., AND DRUSCHEL, P. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2* (2005), USENIX Association, pp. 143–158.

[16] HALALAI, R., FELBER, P., KERMARREC, A.-M., AND TAÏANI, F. Agar: A caching system for erasure-coded data. In *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on* (2017), IEEE, pp. 23–33.

[17] HOSE, K., AND SCHENKEL, R. Warp: Workload-aware replication and partitioning for rdf. In *Data Engineering Workshops (ICDEW), 2013 IEEE 29th International Conference on* (2013), IEEE, pp. 1–6.

[18] HU, Y., AND NIU, D. Reducing access latency in erasure coded cloud storage with local block migration. In *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications* (April 2016), pp. 1–9.

[19] HUANG, C., SIMITCI, H., XU, Y., OGUS, A., CALDER, B., GOPALAN, P., LI, J., AND YEKHANIN, S. Erasure coding in windows azure storage. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)* (2012), pp. 15–26.

[20] HUANG, J., AND ABADI, D. J. Leopard: lightweight edge-oriented partitioning and replication for dynamic graphs. *Proceedings of the VLDB Endowment 9*, 7 (2016), 540–551.

[21] HUANG, Q., BIRMAN, K., VAN RENESSE, R., LLOYD, W., KUMAR, S., AND LI, H. C. An analysis of facebook photo caching. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), ACM, pp. 167–181.

[22] KHAN, O., BURNS, R., PLANK, J., PIERCE, W., AND HUANG, C. Rethinking erasure codes for cloud file systems: minimizing i/o for recovery and degraded reads. In *Proceedings of the 10th USENIX conference on File and Storage Technologies* (2012), USENIX Association, pp. 20–20.

[23] KUBIATOWICZ, J., BINDEL, D., CHEN, Y., CZERWINSKI, S., EATON, P., GEELS, D., GUMMADI, R., RHEA, S., WEATHERSPOON, H., WEIMER, W., ET AL. Oceanstore: An architecture for global-scale persistent storage. *ACM Sigplan Notices 35*, 11 (2000), 190–201.

[24] LAKSHMAN, A., AND MALIK, P. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review 44*, 2 (2010), 35–40.

[25] LI, J., AND LI, B. Zebra: Demand-aware erasure coding for distributed storage systems. In *2016 IEEE/ACM 24th International Symposium on Quality of Service (IWQoS)* (June 2016), pp. 1–10.

[26] LI, J., SHARMA, N. K., PORTS, D. R., AND GRIBBLE, S. D. Tales of the tail: Hardware, os, and application-level sources of tail latency. In *Proceedings of the ACM Symposium on Cloud Computing* (2014), ACM, pp. 1–14.

[27] LI, R., HU, Y., AND LEE, P. P. Enabling efficient and reliable transition from replication to erasure coding for clustered file systems. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks* (2015), IEEE, pp. 148–159.

[28] LI, Z., CHEN, Z., SRINIVASAN, S. M., AND ZHOU, Y. C-miner: mining block correlations in storage systems. In *Proceedings of the 3rd USENIX conference on File and storage technologies* (2004), USENIX Association, pp. 13–13.

[29] MITRA, S., PANTA, R., RA, M.-R., AND BAGCHI, S. Partial-parallel-repair (ppr): a distributed technique for repairing erasure coded storage. In *Proceedings of the Eleventh European Conference on Computer Systems* (2016), p. 30.

[30] MURALIDHAR, S., LLOYD, W., ROY, S., HILL, C., LIN, E., LIU, W., PAN, S., SHANKAR, S., SIVAKUMAR, V., TANG, L., ET AL. f4: Facebook's warm blob storage system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (2014), pp. 383–398.

[31] NISHTALA, R., FUGAL, H., GRIMM, S., KWIATKOWSKI, M., LEE, H., LI, H. C., MCELROY, R., PALECZNY, M., PEEK, D., SAAB, P., ET AL. Scaling memcache at facebook. In *NSDI* (2013).

[32] OVSIANNIKOV, M., RUS, S., REEVES, D., SUTTER, P., RAO, S., AND KELLY, J. The quantcast file system. *Proceedings of the VLDB Endowment 6*, 11 (2013), 1092–1101.

[33] PAIVA, J., RUIVO, P., ROMANO, P., AND RODRIGUES, L. Auto placer: Scalable self-tuning data placement in distributed key-value stores. *ACM Transactions on Autonomous and Adaptive Systems (TAAS) 9*, 4 (2015), 19.

[34] PLANK, J. S., AND GREENAN, K. M. Jerasure: A library in c facilitating erasure coding for storage applications–version 2.0. Tech. rep., Technical Report UT-EECS-14-721, University of Tennessee, 2014.

[35] QUAMAR, A., KUMAR, K. A., AND DESHPANDE, A. Sword: scalable workload-aware data placement for transactional workloads. In *Proceedings of the 16th International Conference on Extending Database Technology* (2013), ACM, pp. 430–441.

[36] RASHMI, K., NAKKIRAN, P., WANG, J., SHAH, N. B., AND RAMCHANDRAN, K. Having your cake and eating it too: Jointly optimal erasure codes for i/o, storage, and network-bandwidth. In *13th USENIX Conference on File and Storage Technologies (FAST 15)* (2015), pp. 81–94.

[37] RASHMI, K., SHAH, N. B., AND RAMCHANDRAN, K. A piggybacking design framework for read-and download-efficient distributed storage codes. In *Information Theory Proceedings (ISIT), 2013 IEEE International Symposium on* (2013), IEEE, pp. 331–335.

[38] RASHMI, K. V., CHOWDHURY, M., KOSAIAN, J., STOICA, I., AND RAMCHANDRAN, K. Ec-cache: Load-balanced, low-latency cluster caching with online erasure coding. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (Savannah, GA, Nov. 2016), USENIX Association, pp. 401–417.

[39] REDA, W., CANINI, M., SURESH, L., KOSTIĆ, D., AND BRAITHWAITE, S. Rein: Taming Tail Latency in Key-Value Stores via Multiget Scheduling. In *Proceedings of the 7th ACM european conference on Computer Systems (EuroSys'17)* (Apr 2017).

[40] REED, I. S., AND SOLOMON, G. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics 8*, 2 (1960), 300–304.

[41] SHAH, N. B., LEE, K., AND RAMCHANDRAN, K. The mds queue: Analysing the latency performance of erasure codes. In *Information Theory (ISIT), 2014 IEEE International Symposium on* (2014), IEEE, pp. 861–865.

[42] SHAH, N. B., RASHMI, K., KUMAR, P. V., AND RAMCHANDRAN, K. Distributed storage codes with repair-by-transfer and nonachievability of interior points on the storage-bandwidth tradeoff. *IEEE Transactions on Information Theory 58*, 3 (2012), 1837–1852.

[43] SHANKAR, D., LU, X., AND PANDA, D. K. High-performance and resilient key-value store with online erasure coding for big data workloads. In *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on* (2017), IEEE, pp. 527–537.

[44] SLEE, M., AGARWAL, A., AND KWIATKOWSKI, M. Thrift: Scalable cross-language services implementation. White Paper: https://thrift.apache.org/static/files/thrift-20070401.pdf, 2007.

[45] SU, M., ZHANG, L., WU, Y., CHEN, K., AND LI, K. Systematic data placement optimization in multi-cloud storage for complex requirements. *IEEE Transactions on Computers 65*, 6 (2016), 1964–1977.

[46] SURESH, P. L., CANINI, M., SCHMID, S., AND FELDMANN, A. C3: Cutting tail latency in cloud data stores via adaptive replica selection. In *NSDI* (2015), pp. 513–527.

[47] URDANETA, G., PIERRE, G., AND VAN STEEN, M. Wikipedia workload analysis. *Vrije Universiteit, Amsterdam, The Netherlands, Tech. Rep. IR-CS-041, Sepember* (2007).

[48] VAJHA, M., RAMKUMAR, V., PURANIK, B., KINI, G., LOBO, E., SASIDHARAN, B., KUMAR, P. V., BARG, A., YE, M., NARAYANAMURTHY, S., HUSSAIN, S., AND NANDI, S. Clay codes: Moulding MDS codes to yield an MSR code. In *16th USENIX Conference on File and Storage Technologies (FAST 18)* (Oakland, CA, 2018), USENIX Association, pp. 139–154.

[49] VENKATARAMAN, S., PANDA, A., ANANTHANARAYANAN, G., FRANKLIN, M. J., AND STOICA, I. The power of choice in data-aware cluster scheduling. In *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation* (2014), USENIX Association, pp. 301–316.

[50] WANG, S., HUANG, J., QIN, X., CAO, Q., AND XIE, C. Wps: A workload-aware placement scheme for erasure-coded in-memory stores. In *Networking, Architecture, and Storage (NAS), 2017 International Conference on* (2017), IEEE, pp. 1–10.

[51] WEATHERSPOON, H., AND KUBIATOWICZ, J. D. Erasure coding vs. replication: A quantitative comparison. In *International Workshop on Peer-to-Peer Systems* (2002), Springer, pp. 328–337.

[52] WEIL, S. A., BRANDT, S. A., MILLER, E. L., LONG, D. D., AND MALTZAHN, C. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation* (2006), USENIX Association, pp. 307–320.

[53] WU, Z., YU, C., AND MADHYASTHA, H. V. Costlo: Cost-effective redundancy for lower latency variance on cloud storage services. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)* (Oakland, CA, 2015), USENIX Association, pp. 543–557.

[54] XIA, M., SAXENA, M., BLAUM, M., AND PEASE, D. A tale of two erasure codes in hdfs. In *FAST* (2015), pp. 213–226.

[55] XIANG, Y., LAN, T., AGGARWAL, V., AND CHEN, Y.-F. Optimizing differentiated latency in multi-tenant, erasure-coded storage. *IEEE Transactions on Network and Service Management 14*, 1 (2017), 204–216.

[56] YIU, M. M., CHAN, H. H., AND LEE, P. P. Erasure coding for small objects in in-memory kv storage. In *Proceedings of the 10th ACM International Systems and Storage Conference* (2017), ACM, p. 14.

[57] ZHANG, H., DONG, M., AND CHEN, H. Efficient and available in-memory kv-store with hybrid erasure coding and replication. In *14th USENIX Conference on File and Storage Technologies (FAST 16)* (2016), pp. 167–180.

[58] ZHU, Y., LIN, J., LEE, P. P., AND XU, Y. Boosting degraded reads in heterogeneous erasure-coded storage systems. *IEEE Transactions on Computers 64*, 8 (2015), 2145–2157.