Latency Reduction and Load Balancing in Coded Storage Systems

Yaochen Hu
University of Alberta
Edmonton, Canada
yaochen@ualberta.ca

Yushi Wang
University of Alberta
Edmonton, Canada
wyushi@ualberta.ca

Bang Liu
University of Alberta
Edmonton, Canada
bang3@ualberta.ca

Di Niu
University of Alberta
Edmonton, Canada
dniu@ualberta.ca

Cheng Huang
Microsoft Research
Redmond, Washington
cheng.huang@microsoft.com

ABSTRACT

Erasure coding has been used in storage systems to enhance data durability at a lower storage overhead. However, these systems suffer from long access latency tails due to a lack of flexible load balancing mechanisms and passively launched degraded reads when the original storage node of the requested data becomes a hotspot. We provide a new perspective to load balancing in coded storage systems by proactively and intelligently launching degraded reads and propose a variety of schemes to make optimal decisions either per request or across requests statistically. Experiments on a 98-machine cluster based on the request traces of 12 million objects collected from Windows Azure Storage (WAS) show that our schemes can reduce the median latency by 44.7% and the 95th-percentile *tail latency* by 77.8% in coded storage systems.

CCS CONCEPTS

• **Networks** → **Network resources allocation**; *Data center networks*; Cloud computing; • **General and reference** → *General conference proceedings*; *Metrics*; *Performance*;

KEYWORDS

Load Balancing, Tail Latency Reduction, Erasure Coded System, Optimization

ACM Reference Format:
Yaochen Hu, Yushi Wang, Bang Liu, Di Niu, and Cheng Huang. 2017. Latency Reduction and Load Balancing in Coded Storage Systems. In *Proceedings of SoCC '17, Santa Clara, CA, USA, September 24–27, 2017,* 13 pages.
https://doi.org/10.1145/3127479.3131623

1 INTRODUCTION

Cloud storage systems, such as Hadoop Distributed File System (HDFS) [3], Google File System (GFS) [8], Windows Azure Storage (WAS)[4] store huge amounts of data that are regularly accessed by personal users and enterprises. Built upon commodity hardware in datacenters, these systems may suffer from frequent data unavailability due to hardware failure, software glitches, I/O hotspots or local congestions. While the first generation of cloud storage systems rely on replication, e.g., 3-way replication in HDFS, for fault tolerance, many current production systems, e.g., Windows Azure Storage (WAS), Google's ColossusFS, Facebook 's HDFS, have adopted erasure coding, e.g., a (k, r) Reed-Solomon (RS) code, to offer much higher reliability than replication at a lower storage cost [11, 26]. Local Reconstruction Codes (LRC) [9, 20, 25] can further reduce the recovery cost, while still maintaining a low storage overhead.

However, the superior durability of erasure coded storage does not come without any tradeoff; it has been widely reported that coded storage suffers from long access latencies [6, 9, 17, 18, 27]. Major cloud providers including Amazon, Microsoft and Google have made a common observation that a slight increase in overall data access latency (e.g., by only 400 ms) may lead to observable fewer accesses from users and thus significant potential revenue loss [21]. Due to the latency issue, coded storage is only limited to storing data that are seldom accessed, and may suffer from long *tail* latencies. Most prior studies have attributed the cause of latency tails to *degraded reads*, defined as the passive action to read multiple (coded) objects from other storage nodes to reconstruct the unavailable object when the original object is temporarily unavailable. Therefore, most studies have focused on inventing new coding schemes to reduce the reconstruction cost, i.e., the number of reads required during degraded reads [9, 11, 17, 20, 25, 27].

In spite of the efforts to reduce recovery cost following data unavailability events, an equally important yet largely overlooked question is—what is the most significant triggering factor of degraded reads in the first place? To answer this question, it is worth noting that most storage systems based on erasure codes today, including Google's ColossusFS and WAS, adopt systematic codes, which place each original uncoded object on a *single* node [11]. A request for an object is first served by the *normal read* from its single original copy, while a timeout can trigger the degraded read. Although such a design choice attempts to avoid degraded reads as much as possible, yet it may not fulfill its intention; by presumably serving every request with the single original object, this design may greatly increase the risk of a storage node becoming congested, forcing degraded reads to be triggered involuntarily [9, 11]. The local congestion issue is further exacerbated by the fact that most real-world demands are highly skewed [6, 18, 27]. In this case, some node hosting hot data may become a hotspot and there is little chance of load balancing in the current system design, since the original copy of each object is stored only on a single node. In contrast, 3-way replication is unlikely to suffer from the same congestion issue, since it can always direct a request to a least loaded node out of the three nodes, each storing a copy of the requested object.

In this paper, we take a radically different approach to latency reduction in coded storage systems. Instead of triggering degraded reads passively following normal read timeouts, we propose to *proactively and intelligently* launch degraded reads in order to shift loads away from hotspots and prevent potential congestion early. Note that we do not consider object write/update, since many big data stores today are append-only, in which each object is immutable and any changes are recorded as separate timestamped objects that get stored on new nodes. Intuitively speaking, if a hot object is attracting a large number of requests which may potentially congest its original storage node, we may serve some of these requests through degraded reads in the first place, without waiting for normal read timeouts. Although proactive degraded reads may reduce the longest queues, they may flood the system with more reading tasks and affect the service latencies for other requests in general. Therefore, we must carefully decide: 1) for which request a degraded read should be performed, and 2) should a degraded read be used, from which storage nodes the degraded read should be served.

Toward these objectives, we propose a variety of load balancing approaches to reduce latencies in erasure coded storage systems, including statistical optimization that can globally coordinate different requests and per-request optimal decisions. A first approach is an efficient optimization framework that intelligently maintains a load direction table between all requests and storage nodes, based on periodically sampled demand and queue statistics. This approach is sub-optimal since it only updates direction decisions periodically, failing to utilize instantaneous load information. We then naturally turn to per-request optimal decisions, one of which is *least latency first*, that is to serve each request with the normal read or a degraded read, whichever minimizes the current request latency. However, this may lead to an excessive number of degraded reads and increase overall system burden, affecting future requests. To solve this issue, we introduce the key notion of *marginal load* and propose a novel *least-marginal-load-first* policy which judiciously and lazily launches degraded reads for load balancing based on server queue length statistics, without flooding the system. To reduce the server queue length probing overhead, we further adapt the power-of-two sampling idea to our per-request optimal load balancing in coded storage systems. We show that the per-request optimal decision is essentially the optimal solution to the statistical optimization problem for each single request with a specific objective function. Note that per-request optimal decisions have an inherent probing overhead that scales with the demand, such that a large number of concurrent controllers must be used for heavy workloads. In contrast, the statistical optimization, though being a sub-optimal approach, is scalable since it only needs to make a small fixed amount of probes in each period.

We deployed a coded storage testbed on 98 machines to evaluate the performance of the proposed schemes by replaying a large amount of real request traces collected from Windows Azure Storage. Results suggest that the proposed schemes based on proactive degraded reads can reduce the median latency by more than 40% and the 95-percentile tail latency by more than 75% in RS-coded systems and LRC-based systems, as compared to the current approach of normal read with timeouts. We show that *least marginal load first* can achieve supreme latency reduction when there is an enough number of controllers and the network I/O is not a bottleneck, whereas the statistical optimization can yield a latency close to *least marginal load first* with inertia probing, yet achieving a higher request processing throughput when the number of controllers is limited.

2 LOAD BALANCING IN CODED STORAGE

In the traditional design of erasure coded storage systems, degraded reads are triggered passively to serve a request when the *storage node* (server) storing the original requested object is temporarily unavailable or to restore a failed server. We take a radically different approach, by letting the system intentionally and intelligently perform degraded reads based on demand information and load statistics in the system to direct requests away from hot servers. Bearing request latencies, server load balancing and network I/O overhead in mind, we present several approaches to decide for which request a degraded read should be launched and from which servers the degraded read should be served.

2.1 Terminology and System Model

A cloud storage cluster makes redundant copies of each single logic unit of data in order to maintain the availability of highly demanded data. In such systems, a large number of small *objects* are grouped to form relatively larger *partitions* (or blocks), typically each of size 64 MB to several GBs. In a *replication-based system*, these partitions are replicated and placed on several different storage nodes. Each incoming request is served by one of the replicas [13] chosen either randomly or according to more sophisticated load balancing schemes. Such systems suffer from a high storage overhead. For the typical 3-replica, there is a 3× storage overhead.

Erasure coded systems are now widely adopted to reduce storage overhead while achieving high reliability and availability. Partitions form *coding groups* (or stripes). In each group, *parity partitions* are generated from the original partitions. The original and parity partitions are spread across different storage nodes. For instance, with a typical $(6, 3)$ RS code, in each coding group, 3 parity partitions are generated from 6 original partitions. Each of the 9 partitions can be recovered from any other 6 partitions in the same coding group. A $(6, 3)$ RS code can reduce the storage overhead down to 1.5× with a higher reliability than 3-replica.

Each request is usually directed to the node that stores the original partition containing the requested object. We call this storage node the *original node* for the object and a read from the original node a *normal read*. When the normal read has a large delay due to temporal unavailability of the corresponding storage node, the request will be served by a *degraded read*, that is to read any 6 other partitions in the same coding group. In both a normal read and a degraded read, we do not need to read the entire partition(s); only the *offset* corresponding to the requested object needs to be read from each partition. A common problem of RS coded storage is that the system will suffer from high *recovery cost* defined as the number reads that must be performed to recover an unavailable object. Other codes have been proposed to further reduce the recovery cost, e.g., the Local Reconstruction Code (LRC) [9] optimizes the recovery cost for the failure or unavailability of a single node, which is a common case in practice. Specifically, for a $(6, 2, 2)$ LRC, every 6 original partitions form a coding group, divided into two subgroups. One local parity partition is generated for each subgroup and there are two global parity partitions. Every single node failure can be recovered from 3 partitions in the local subgroup. The failure of 3 partitions in the same subgroup and some failures of 4 partitions can be recovered with the help of global parity partitions.

For a typical cloud storage system, such as WAS, client requests first arrive at certain *frontend servers* or gateways. The frontend servers direct the incoming requests to different storage nodes subject to content placement constraints and certain load balancing policies. Requests are then directly served by the selected storage node(s) to the clients. We use *request latency* to describe the time gap from the arrival of a request at the frontend server until the request is fully served, and use *task latency* to describe the time that it takes to perform a read task for a particular single object (coded or uncoded) being assigned to some storage node. For example, in a $(6, 3)$ RS coded system, the request latency for a normal read is just the queuing time of the request at the frontend server plus the task latency of a single read. In contrast, a degraded read will launch 6 read tasks on 6 different storage nodes. In this case, the request latency will be the longest read task latency plus the queuing time of the request at the frontend server (gateway).

We can assume that the storage nodes are homogeneous in terms of network configuration and task processing speeds, which is common in pratice. However, our ideas can easily be generalized to heterogeneous storage nodes by considering the processing speeds of servers.

2.2 Proactive Degraded Reads

In a replication-based system, each request can be directed to the least loaded storage node storing the requested object. However, for an erasure coded system, the original object is stored on only one storage node with little opportunities for load balancing. Therefore, traditionally, degraded reads are launched only when the normal read has timed out. However, in this paper, we show that launching degraded reads proactively for carefully selected requests can in fact reduce access latencies and improve the overall system efficiency.

From the request traces of WAS, we found that the requests are highly skewed: most requests are for a small

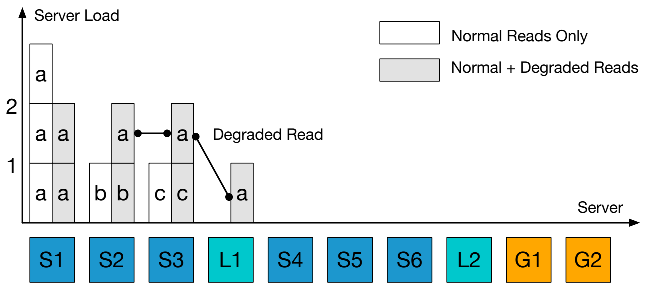


Figure 1: Illustration on why carefully triggered degraded reads may help balance the loads.

portion of partitions. Consider a toy example of (6,2,2) LRC code in Fig. 1, where 3 original partitions placed on servers $S1$, $S2$ and $S3$ form a subgroup, with a local parity partition stored on server $L1$. Suppose that server $S1$ is already heavily loaded with requests, while $S2$, $S3$ and $L1$ are relatively less loaded. When a request for an object on sever $S1$ comes, the traditional scheme still directs the request to $S1$, since $S1$ is still available although heavily loaded. In contrast, if we have proactively launched a degraded read (on servers $S2$, $S3$ and $L1$ together) to reconstruct the requested object in the first place, both the request latency and the load of server $S1$ can be reduced.

2.3 Statistical Optimization

To answer the question whether to serve a request with a normal read or a degraded read and which servers to serve the degraded read, we first inspect the problem in an optimization point of view. Other approaches are inspired and justified by the optimization framework.

We adopt a statistical optimization approach based on the queue status refreshed periodically as well as the request history in near past. Specifically, we keep a *load direction table* computed periodically by solving an optimization problem to be described soon, based on the latest storage node queue status and request statistics measured within a certain window. The table specifies optimized proportions at which the requests for an object in each partition should be served by the normal read or each degraded read combination. For each non-splittable request that arrives at a frontend server, load direction is made at random according to the probabilities derived from the proportions given in the load direction table.

Although we direct the load via the optimal solution, the result is still sub-optimal since the table is only updated periodically, failing to utilize the instantaneous load information. However, this approach only needs a fixed amount of probes in each period and thus is scalable to the number of requests. We present the details of the statistical optimization model in the remaining subsection.

Suppose that n partitions are placed on m storage nodes. To take into account the pending tasks on each node, let $\vec{Q} = (Q_1, \ldots, Q_m)$ represent the existing queue sizes (in bytes) on m storage nodes. Denote by $s(i)$ the original node of partition i. Let the k-tuple $c = (s_{j_1}, \ldots s_{j_k})$ denote a combination of k storage nodes, and C_i be the set of all k-node combinations c which can serve degraded reads for objects in partition i. Note that for an RS code, k is fixed. For an LRC, k could take several values depending on whether a local or global recovery is triggered [4].

Suppose D_i is the instantaneous total request size per second for objects in partition i, $i = 1, \ldots, n$. For each request for an object in partition i, we use $x_i \in [0, 1]$ to denote the probability of serving it with the normal read and $y_{ic} \in [0, 1]$ to denote the probability of serving it with a degraded read through the combination $c \in C_i$. Apparently, we must have $x_i + \sum_{c \in C_i} y_{ic} = 1$. Let $\vec{L} = (L_1, \ldots, L_m)$, where L_j is the *expected load* of storage node j as a result of load direction. Then, due to the linearity of expectations, the expected load L_j is a weighted sum of all the demands $D_1, \ldots, D_n$ weighted by the load direction probabilities x_i and $\{y_{ic} | c \in C_i\}$, $i = 1, \ldots, n$.

Let $F(\cdot)$ be a *load balancing metric* defined as a function of $\vec{L} + \vec{Q}$, i.e., the expected loads $\vec{L}$ on all storage nodes plus the existing queues pending on them. Therefore, the optimal load direction probabilities are the solution to the following problem:

$$\underset{\{x_i\},\{y_{ic}\}}{\text{minimize}} \quad F(\vec{L} + \vec{Q}) \tag{1}$$

$$\text{subject to} \quad L_j = \sum_{\{i:s(i)=j\}} D_i x_i + \sum_{\{(i,c):j \in c, c \in C_i\}} D_i y_{ic}, \quad j = 1, \ldots, m, \tag{2}$$

$$x_i + \sum_{c \in C_i} y_{ic} = 1, \quad i = 1, \ldots, n. \tag{3}$$

$$y_{ic} \geq 0, \quad \forall c \in C_i, \; i = 1, \ldots, n, \tag{4}$$

$$x_i \geq 0, \; i = 1, \ldots, n, \tag{5}$$

where load L_j directed to storage node j is given by (2). The first term in (2) is the sum of all normal reads $D_i x_i$ with the storage node j being the original node, while the second term is the sum of all degraded reads $D_i y_{ic}$ that may incur loads on node j, i.e., $j \in c, c \in C_i$.

Several load balancing metrics can be used, for example, the ℓ_∞ norm: $F(\vec{L}+\vec{Q}) = \|\vec{L}+\vec{Q}\|_\infty = \max_{j=1,\ldots,m}(L_j + Q_j)$, which models the longest expected queue length after load direction, and the ℓ_2 norm: $F(\vec{L} + \vec{Q}) = \frac{1}{2}\|\vec{L} + \vec{Q}\|_2^2 = \frac{1}{2}\sum_{j=1}^{m}(L_j + Q_j)^2$, which models the aggregated per-byte processing latency in the system, assuming a similar processing rate across storage nodes in a homogeneous system. The metric can easily accommodate the case of heterogeneous systems where storage node j has a data processing rate of S_j; in this case, $F(\vec{L}+\vec{Q}) = \frac{1}{2}\sum_{j=1}^{m}(L_j + Q_j)^2/S_j$. If the ℓ_∞ norm is used, problem (1) is a linear program, and if the ℓ_2 norm is used, problem (1) is a quadratic program. In our experiment, we adopt the ℓ_2 norm. Standard tools like MOSEK[2] can solve it efficiently with worst-case time complexity of $O(n^3)$.

2.4 Per-Request Optimal Decisions

With the statistical optimization approach, the load direction table is updated in a synchronized manner and is not changed within each cycle, leading to a sub-optimal result. A more direct approach is to instantaneously probe the queue status of related data nodes and to make an optimal decision for each request. In this case, we need a criterion to measure how good a load direction choice is. Optimal policies can be derived by adapting problem (1) to per-request decisions under different load balancing metrics.

Least Latency First. We first introduce the *least latency first* policy, which corresponds to the per-request minimization of the ℓ_∞ norm of storage node queue sizes (or the maximum queue size). Consider a single request for an object of size D in partition i_0. In this case, the optimal policy comes from a solution to a special case of problem (1), with $D_{i_0} = D > 0$ and $D_i = 0, \forall i \neq i_0$, that is to solve problem (1) for this single request. Every single non-splittable request will be served by either the normal read or one of the degraded combinations. The optimal choice will be the one that leads to the smallest objective value in problem (1).

To find out the optimal decision, we can choose either the normal read or a degraded read that results in the lowest estimated request latency for the current request. For the normal read, the request latency almost equals to the corresponding task latency, which can be estimated *by the total size of all queued requests at the storage node plus the requested object size, divided by the average node processing rate.* For a degraded read served by k other storage nodes, the request latency can be estimated *by the longest task latency among the k nodes.* Note that in problem (1), different servers are assumed to have the same processing rate. Thus, the processing rate is omitted in the objective function.

For example, the least-latency-first policy can be performed in a $(6,3)$ RS coded system in the following way: Upon the arrival of a request, the queue sizes at the 9 storage nodes that can serve this request are probed, including the original node and 8 other nodes, any 6 of which can serve the request via a degraded read. Then, we will compare the task latency of the normal read with the longest task latency among the 6 *least loaded nodes* out of the other 8 nodes, and pick whichever is smaller to serve the request.

However, the least-latency-first policy may not perform the best in the long term since it only optimizes the latency of *the current request in question* regardless of future requests. In fact, for a request for a hot object, the least-latency-first policy tends to shift load away from the original node, whenever there are at least 6 other nodes in the same coding group with a queue size smaller than that of the original node. Although such a "water-filling" approach will help to balance server loads, the drawback is that it encourages degraded reads too much and increases the overall number of read tasks launched in the system, which may prolong the service latencies of future requests, as their original nodes have been used to perform degraded reads for earlier requests. Therefore, we need a load direction policy to reduce the burden of heavily loaded storage nodes, while still penalizing degraded reads.

Least Marginal Load First. To strike a balance between reducing the current request latency and minimizing the overall system load, we propose a different metric to measure different load direction choices. We introduce the *least-marginal-load-first* policy. Similar to the case of LLF, this policy is essentially an optimal solution to problem (1), but with an ℓ_2 objective function. Let us consider the special case of problem (1) again, with $D_{i_0} = D > 0$ and $D_i = 0, \forall i \neq i_0$, which is to solve problem (1) for this single request. Comparing the ℓ_2 objective function values before and after the request is directed, each direction decision will increase the objective by a certain amount. Specifically, for the normal read, the increase is

$$\begin{aligned}\Delta F_{s(i_0)} &= \frac{1}{2}\sum_{j\neq s(i_0)} Q_j^2 + \frac{1}{2}(Q_{s(i_0)} + D_{i_0})^2 - \frac{1}{2}\sum_j Q_j^2 \\ &= D_{i_0}(Q_{s(i_0)} + \frac{1}{2}D_{i_0}) = D(Q_{s(i_0)} + \frac{1}{2}D), \qquad (6)\end{aligned}$$

and for a degraded read with $c \in C(i_0)$, the increase is

$$\begin{aligned}\Delta F_c &= \frac{1}{2}\sum_{j\notin c} Q_j^2 + \frac{1}{2}\sum_{j\in c}(Q_j + D_{i_0})^2 - \frac{1}{2}\sum_j Q_j^2 \\ &= \sum_{j\in c} D_{i_0}(Q_j + \frac{1}{2}D_{i_0}) = \sum_{j\in c} D(Q_j + \frac{1}{2}D). \qquad (7)\end{aligned}$$

The optimal choice would be the one that leads to the minimum increase of the objective function. We can pick it out by computing the value of (6) for the normal read and the values of (7) for all degraded reads and selecting the one with the minimum value.

Intuitively speaking, consider assigning a read task of size D on a storage node with existing queue size Q. With a similar processing rate across servers, $Q + D/2$ can be regarded as the per-byte *average* processing time.

$D \cdot (Q + D/2)$ is the summation of processing times per byte in this read task. We call this value the *marginal load* of the read task of size D. Let us now define the marginal load of a normal read and that of a degraded read. The marginal load of a normal read is just the marginal load of the corresponding single read task, whereas the marginal load of a degraded read is *the summation of all the marginal loads of the associated read tasks.* We define the *least-marginal-load-first* (LMLF) policy as choosing the normal read or a degraded read that achieves the least marginal load, which is naturally an optimal solution to problem (1) for a single request with an ℓ_2 objective function.

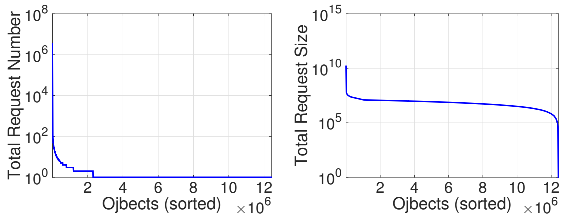
For example, in a $(6, 3)$ RS coded system, for an object request, *least marginal load first* will compare the marginal load $DQ_0 + D^2/2$, where Q_0 is the original node queue size, with $\sum_{i=1}^{6}(DQ_i + D^2/2)$, where $Q_1, \ldots, Q_6$ are the queue sizes of the 6 *least loaded nodes* out of the other 8 nodes, and pick whichever is smaller to serve the request.

LMLF strikes a balance between reducing the current request latency and minimizing the overall system load. On one hand, storage nodes with a large Q incur a larger marginal load and are less likely to be selected. On the other hand, the marginal load of a degraded read is the summation of the marginal loads on all associated nodes, penalizing degraded reads from flooding the system. Moreover, objects with a larger size D are less likely to be served by degraded reads due to their higher marginal loads, attributed to the additive term $D^2/2$ in the marginal load of each associated task. In other words, LMLF launches degraded reads lazily and saves the system resources for future requests.

2.5 Distributed Power-of-Two Sampling

The per-request optimal decisions above rely on instantaneously probed queue sizes for each request, incurring much probing overhead at the frontend (gateway) servers. For a (k, r) RS code, $k + r$ storage nodes need to be probed for each request.

To save the probing overhead, we exploit an idea similar to the Power of Two in job scheduling for load direction in coded storage systems. Specifically, we can just pick a random degraded read and only probe the k storage nodes related to this degraded read and compare its marginal load with that of the normal read. Taking the $(6, 3)$ RS code as an example, instead of having 9 probes for each request, in the sampled solution, we only need to probe the original node and a set of randomly chosen 6 storage nodes in the same coding group as the requested object, and use whichever achieves a lower marginal load, which saves the probing overhead by 22.2%. In particular, codes that are designed to optimize the recovery cost for single node failures will benefit the most from sampling. In a typical $(6, 2, 2)$ LRC, if only the local recovery combination and the normal read are considered, we only need to probe 4 storage nodes for each request, which saves the probing overhead by 60%, compared to the full probing of 10 storage nodes.



(a) Distribution of aggregated request number (b) Distribution of aggregated request size (bytes)

Figure 2: The properties of the trace data collected from Windows Azure Storage (WAS).

Theoretical results [13] on Power-of-Two load balancing in traditional queuing systems have shown that the expected performance will not drop too much as compared to full probing. We will verify the effectiveness of our version of distributed power-of-two sampling with experiments.

2.6 Summary

We propose different methods to help making the optimal decision for the proactive degraded reads. There is a tradeoff in each method. Least latency first probes queue status and optimizes for each request instantaneously. But it does not coordinate different requests and incurs larger probing overhead. Least marginal load first not only optimizes for each request with instantaneous probing, but also saves system resources for future requests by penalizing degraded reads. The distributed power-of-two sampling can alleviate the probing burden at the cost of a slight deviation from the optimal solution. Finally, in the case that probing overhead could form a bottleneck, statistical optimization can be used to jointly direct the loads for all requests taking advantage of joint statistics of different requests, although the solution is only an approximation to the optimality due to the lazily updated demands and queue table.

3 IMPLEMENTATION AND EXPERIMENT SETUP

We implemented and deployed a prototype coded storage testbed to evaluate the performance the proposed load balancing mechanisms on a cluster of 98 Amazon EC2 virtual machines (which do not use SSD) by replaying the request traces we have collected from Windows Azure Storage (WAS), containing the request logs for

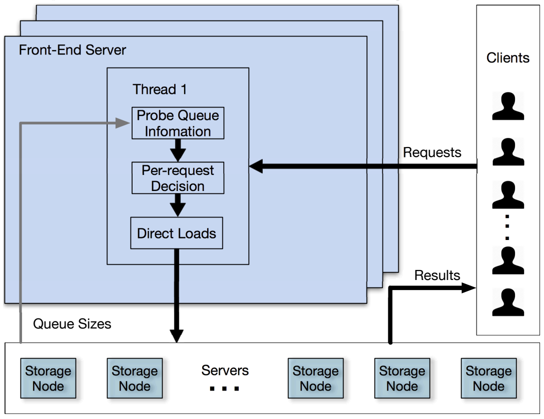


Figure 3: System architecture with per-request optimal decisions based on instantaneous probing.

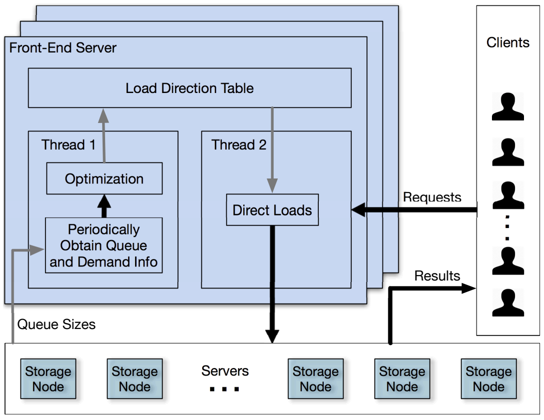


Figure 4: System architecture with statistical optimization based on lazy queue status updates.

12 million objects over a one-day period. The objects are randomly grouped into 2004 partitions of an equal size. Fig. 2 plots the distribution of the aggregated demands in terms of request number and request file size for different objects in the trace. We can see that they are highly skewed: a small number of hot objects attract a large amount of requests, while most objects are cold. We use a randomized partition placement strategy, typically adopted in WAS: when a (k, r) RS code is applied, each coding group has k original and r parity partitions. These $k + r$ partitions are placed on $k + r$ random storage nodes such that the storage cost at each node is balanced; when a (k, l, r) LRC is used, a coding group of $k + l + r$ original/parity partitions are placed on $k + l + r$ random nodes in a similar way.

Our testbed has an architecture shown in Fig. 3 and Fig. 4, consisting of two major parts: *frontend servers* and *storage nodes*. Each frontend server works in a decentralized way, and can receive requests and direct them to storage nodes according to a load balancing policy. Each frontend has access to the content placement map that indicates which partition is stored on which storage nodes, as well as the offsets of each object in its partition. For each request, we just need to read from the specified offset in the corresponding partition for the object size. In our experiments, the requests in the traces (each in a form of <time>, <object id>, <size requested>) are fed to one frontend server at random to mimic the way that requests arrive at frontend servers from the clients in a balanced way. We do not consider more sophisticated frontend load balancing, since the focus is on the load balancing of storage nodes.

The read tasks on each storage node are executed sequentially in a first-come-first-service (FCFS) queue. Each storage node keeps track its queue size. Since the read tasks are different in sizes (due to different object sizes), the number of tasks on a storage node is not a good estimate of its real load [15]. Our system uses the aggregated size of read tasks on each storage node to estimate its queue size, which is also easy to keep track of. We do not consider object write/update, since many big data stores today are append-only, in which each object is immutable and any changes are recorded as separate timestamped objects that get stored on new nodes. Hence, write requests are unlikely to create a hotspot on a single node and our focus here is on enabling faster reads.

3.1 The Controller and Logging

Each frontend server has a controller to execute load direction policies. In this work, we evaluate the performance of the following polices:

1) **Normal**: normal read with degraded read triggered upon timeout (a common existing solution);
2) **LLF**: least latency first with per-request full probing of all related storage nodes;
3) **LMLF**: least marginal load first with per-request full probing of all related storage nodes;
4) **LLF PW2**: least latency first with per-request power-of-two probing on sampled nodes;
5) **LMLF PW2**: least marginal load first with per-request power-of-two probing on sampled nodes;
6) **LMLF Lazy**: least marginal load first with periodic updates of storage node queue status;
7) **Opt Lazy**: statistical optimization of ℓ_2 norm with periodic updates of storage node queue status.

For LLF and LMLF as shown in Fig. 3, the controller probes all the related nodes for each request, e.g., 9 nodes for a $(6, 3)$, for their current queue sizes and make per-request decisions based on LLF or LMLF. For LLF PW2 and LMLF PW2, only the original node and the nodes for one degraded read combination are probed: for RS, it is a random degraded read, and for LRC, it is the local recovery combination. Thus, LLF PW2 and LMLF PW2 have lower probing overhead. For LMLF Lazy, one separate thread probes all queue status periodically and updates the queue table on all controllers every T seconds, while each controller makes an LMLF decision for each incoming request based on the lazily updated queue table.

For Opt Lazy as shown in Fig. 4, besides the queue table updated every T seconds, the aggregate request rate for each partition is also recorded from the previous T seconds. Then, a separate thread, implemented with MOSEK[2] in Python, computes a *load direction table* with the statistical optimization method in Sec. 2.3, and obtains the optimal *probabilities* that a request for an object in each partition will be served by the normal read *or* certain degraded reads. Note that the same load direction probabilities are applied to all the objects in a same partition, thus limiting the load direction table to a practical size. We do not differentiate between the objects in a partition, since overly fine-grained load direction increases overhead and partition-level load balancing is sufficient in our experiments. Each controller then probabilistically directs each incoming request based on the current load direction table.

Note that Opt Lazy only solves problem (1) between the normal read and a single degraded read: for RS, it is a random degraded read, and for LRC, it is the local recovery combination. This simplification significantly reduces the optimization computation time to around 0.1 second for our problem sizes yet achieving similar objective values to full optimization according to simulations. To simplify the synchronization issue among frontend servers, we let each controller compute a separate (yet nearly identical) load direction table every T seconds based on the current queue status and collected demand information. And there are two separate concurrent threads for computing the load direction table and for directing each incoming request based on the current table.

We record the following real timestamps for each request: *A*) the time that the request arrived at the frontend server, *B*) the time that the frontend server sent the read task(s) of this request to storage nodes, *C*) the time that each task entered the queue at the designated storage node, *D*) the time that the task processing started, *E*) the time that the request processing finished.

4 EXPERIMENTAL RESULTS

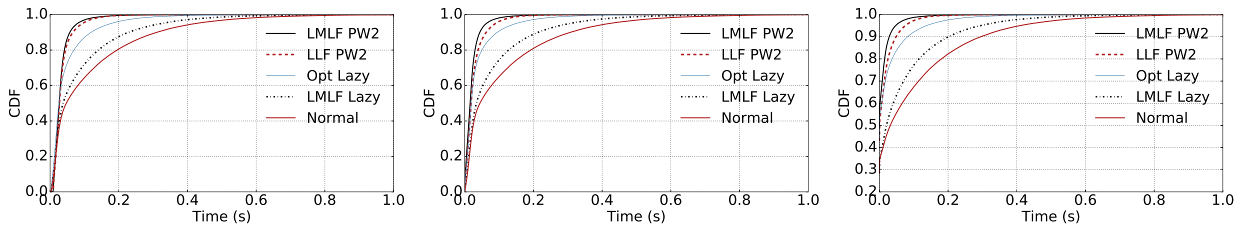
Our evaluation is based on replaying a large amount of traces collected from the Windows Azure Storage (WAS), containing the request logs for 12 million objects over a one-day period. We randomly group them into 2004 partitions of an equal size and import them into the coded storage testbed deployed on a 98-machine cluster, including 90 Amazon EC2 *t2.nano* instances (which are not based on SSD to mimic commodity hardware used in practice), which serve as the storage nodes, and 3 or 8 quad-core Amazon EC2 *m4.xlarge* instances, which serve as the front-end servers. Fig. 2 shows the aggregated request numbers and sizes for different objects. The requests are highly skewed in that a few objects have contributed a considerable portion of the total workload to the system.

We replayed the traces in a typical peak hour under different load direction polices listed in Sec.3, respectively, in the testbed for both a $(6, 3)$ RS coded storage system and a $(6, 2, 2)$ LRC-based storage system, and record the timestamps we mentioned in Sec. 3.1. For the LRC system, we evaluated the following 5 polices: Normal, LLF PW2, LMLF PW2, LMLF Lazy and Opt Lazy, since in LRC probing all the degraded read combinations will incur an overly high cost. For the $(6, 3)$ RS coded system, we evaluated all the polices listed in Sec. 3.1. In our experiment, we set the queue status probing frequency to once per 4 seconds for Opt Lazy and once per second for LMLF Lazy. We use 8 front-end servers for the LLF, LMLF, LLF PW2, LMLF PW2 policies to support a high capability for instantaneous probing and 3 front-end servers for the Normal, Opt Lazy and LMLF Lazy policies, which have much lower probing overhead.

4.1 Performance in the LRC-based System

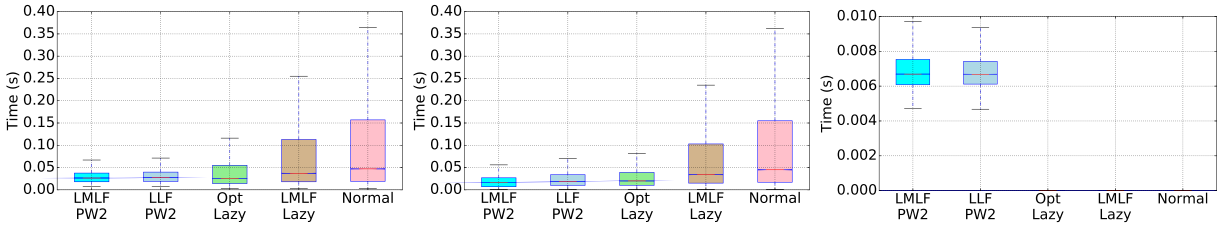
For an LRC-based system, Fig. 5(a) and Fig. 6(a) show the request latencies under different policies. The request latency is the actual latency each request encounters including the controller processing time (the delay at the controller before read tasks are launched) and the maximum of related task latencies. Fig. 5(b) and Fig. 6(b) show the task latencies under different policies. The task latency is the time gap between the time a task is inserted into the queue of a storage node and the completion of the task. Fig. 7(c) shows the task waiting time which is the time between a task entering a storage node and the beginning of its processing. Fig. 6(c) shows the controller processing time, which is the delay each request experiences at the controller of the front-end server, including queuing and decision-making. Table 1 further shows the overall statistics of the performance of different polices.

First, we can see that all the polices with judiciously launched proactive degraded reads can improve the access latency compared to the traditional policy Normal that triggers degraded reads following the timeouts of normal reads. Specifically, LMLF PW2, using the marginal load and power-of-two sampled probes for optimal per-request decisions, reduces the mean latency by



(a) CDF of request latencies (b) CDF of task latencies (c) CDF of task waiting times

Figure 5: The CDFs of different latency metrics for the storage system based on a (6,2,2) LRC.



(a) Box plot of request latencies (b) Box plot of task latencies (c) Box plot of controller processing times

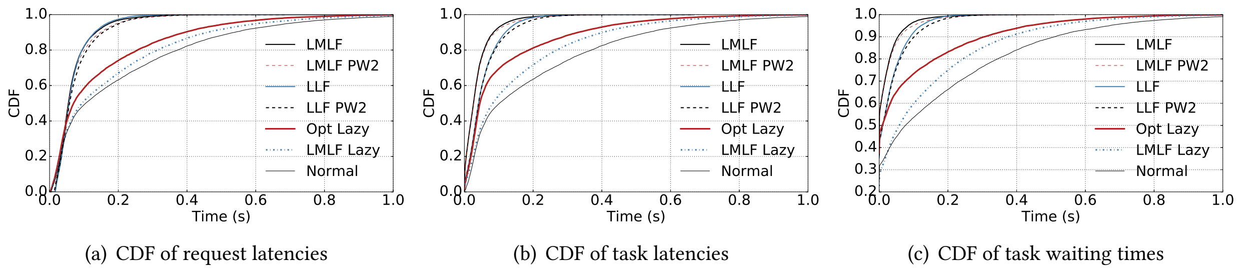
Figure 6: Box plots for different latency metrics for the storage system based on a (6,2,2) LRC.

44.7% and the 95th-percentile by 77.8% with respect to Normal.

Least Marginal Load First vs Least Latency First. Fig. 5(b) and Fig. 6(b) show the task latencies for different policies. LMLF PW2 is considerably better than the LLF PW2 by using the marginal load as a load balancing criterion. As we have analyzed, LMLF PW2 tries to strike a balance between the reduced individual request latency and the overall system load while LLF PW2 only makes decisions to optimize the current request latency, thus launching degraded reads too aggressively. Although LMLF PW2 does not optimize the current request latency directly, it is essentially a per-request optimization for the aggregated task latencies in the long run as we have shown in Sec. 2. In Fig. 5(a) and Fig. 6(a), the advantage of LMLF PW2 over LLF PW2 in terms of the request latency is not as obvious as it is in terms of the task latency. This is due to a higher decision-making complexity on the front-end servers for LMLF PW2 as shown in Fig. 6(c) and the fact that the request latency for the degraded read is dominated by the maximum of all the related task latencies. In general, LMLF PW2 outperforms LLF PW2 with better request latency performance and much better task latency performance.

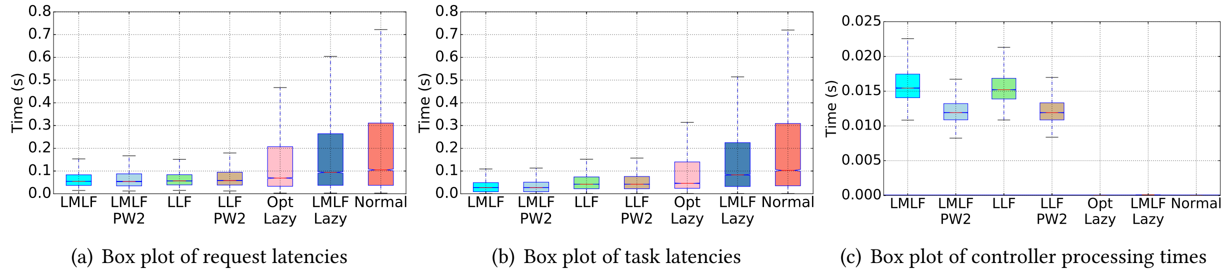
Opt Lazy vs. LMLF PW2. In terms of the request latency and task latency as shown in Fig. 5(a) and Fig. 5(b), LMLF PW2 outperforms the optimization scheme Opt Lazy. However, Opt Lazy needs much fewer probes than the LMLF as is shown in Table 1. In fact, the probing overhead of LMLF PW2 scales linearly as the number of requests increases while for Opt Lazy, the probing overhead is linearly related to the number of storage nodes, thanks to the use of periodic lazy updates. Therefore, Opt Lazy may potentially be able to handle a much larger amount of requests than the LMLF PW2 especially when the number controllers is limited. Moreover, the controller processing time of LMLF PW2 shown in Fig. 6(c) is much larger than that of Opt Lazy. Opt Lazy can efficiently assign the requests to the storage nodes based on the statistical optimization results calculated by another thread while LMLF PW2 might suffer from a possibly long probing delay or even timeout when the demand is too high as compared to the available network bandwidth. This phenomenon is not obvious in our system since the network environment was very stable when the experiments were conducted.

Opt Lazy vs. LMLF Lazy. Both of the two approaches only need to acquire the queue status of the storage node periodically, thus with low probing overhead. However, Opt Lazy outperforms LMLF lazy significantly since LMLF Lazy usually suffers from race conditions since queue status is not up-to-date. The optimization method makes decisions based on both the periodically updated queue status and request statistics, which help to make a better decisions by jointly consider and coordinate the requests for objects in different partitions.



(a) CDF of request latencies (b) CDF of task latencies (c) CDF of task waiting times

Figure 7: CDFs of different latency metrics for the storage system based on a (6,3) RS code.



(a) Box plot of request latencies (b) Box plot of task latencies (c) Box plot of controller processing times

Figure 8: Box plots of different latency metrics for the storage system based on a (6,3) RS code.

4.2 Performance in the RS-coded System

Fig. 7(a) and Fig. 8(a) plot the request latencies under different load direction policies in a (6,3) RS coded system. Fig. 7(b) and Fig. 8(b) plot the task latencies, while Fig. 7(c) shows the task waiting times. Fig. 8(c) shows the box plot for task waiting times and controller processing times. Table 2 further shows the overall statistics of the performance of different polices. For the RS code, the performance demonstrates a similar overall trend as that for the LRC system: we can reduce the request latency significantly by judiciously launching proactive degraded reads.

LMLF PW2 vs. LMLF. As shown in Fig. 7(a) and Fig. 7(b), LMLF is better than LMLF PW2 since it makes per-request optimal choices based on full probing. However, the improvement is limited, since the requests are highly skewed and most of the time, the decision between whether to trigger a degraded read or not has a more significant impact to the performance while selecting which recovery combination for a degraded read is relatively less important. There is a similar trend between LLF PW2 and LLF. Therefore, there is a high motivation to use the LMLF PW2 and LLF PW2 instead of LMLF and LLF with full probing to save the probing cost. Note that in a (6,3) RS coded system, the saving of probes is 1/3. This amount varies among different erasure coded schemes depending on the total number of partitions in a coded group and how many partitions are needed to recover a single partition.

LMLF vs. LLF. As shown in Fig. 7(a), the performance of LMLF seems to be slightly worse than LLF. Similar to the case in LRC system, LMLF has higher controller processing time. Also, LMLF tries to keep the overall workload of the storage nodes to a lower level and only reduce the request latency tail. On the other hand, LLF searches for a decision with minimum per-request latency directly and thus gets a better request latency. But as is shown in Fig. 7(b), LMLF has much better task latencies so that it incurs a much lower overall task workload on the storage system and has the potential to serve more demands.

5 RELATED WORK

Most prior work has attributed the long latency tails in coded storage to degraded reads. A large amount of research has focused on reducing the recovery cost during degraded reads. Local Reconstruction Code (LRC) [9] is proposed to reduce the IOs required for reconstruction over Reed-Solomon (RS) codes, while still achieving significant reduction in storage overhead as compared to 3-way replication. Similar locally recoverable codes have been presented in [20, 25]. HitchHiker [17] is another erasure-coded storage system that reduces both network traffic and disk I/O during reconstruction, residing on

Table 1: Overall request latencies (seconds) and probing overhead (# probes/s) for the LRC-based system.

Policy	mean	var	median	95th	min	max	# probs per second
Normal	0.114	0.022	0.047	0.424	0.003	1.244	0
LMLF Lazy	0.087	0.013	0.037	0.324	0.003	1.197	270
Opt Lazy	0.048	0.005	0.025	0.177	0.003	1.038	67.5
LMLF PW2	0.033	0.001	0.026	0.080	0.008	1.038	888.9
LLF PW2	0.036	0.001	0.027	0.094	0.008	1.040	888.9

Table 2: Overall request latencies (seconds) and probing overhead (# probes/s) for the RS coded system.

Policy	mean	var	median	95th	min	max	# probs per second
Normal	0.207	0.056	0.105	0.695	0.003	1.951	0
LMLF Lazy	0.180	0.041	0.094	0.607	0.003	1.331	270
Opt Lazy	0.148	0.031	0.069	0.520	0.003	1.420	67.5
LMLF PW2	0.073	0.005	0.054	0.196	0.012	3.043	784.5
LLF PW2	0.076	0.004	0.058	0.203	0.012	0.684	784.5
LMLF	0.070	0.003	0.055	0.174	0.014	1.016	1008.7
LLF	0.069	0.002	0.057	0.166	0.015	1.053	1008.7

top of RS codes based on new encoding/decoding techniques. HACFS [27] uses two different erasure codes, i.e., a fast code for frequently accessed data to lower the recovery cost, and a compact code for the majority of data to maintain a low overall storage overhead. [31] presents an algorithm that finds the optimal number of codeword symbols needed for recovery with any XOR-based erasure code and produces recovery schedules to use a minimum amount of data. [11] proposes FastDR, a system that addresses node heterogeneity and exploits I/O parallelism to enhance degraded read performance.

However, another important question is: what is the cause of degraded reads in coded storage in the first place? In fact, aside from node failures, the majority of degraded reads are passively triggered during *temporary unavailability* of the original node [9, 17, 27]. For example, Over 98% of all failure modes in Facebook's data-warehouse and other production HDFS clusters require recovery of a single temporary block failure [16] instead of node failures. And only less than 0.05% of all failures involve three or more blocks simultaneously. Furthermore, a major reason underlying such temporary node unavailability is that under skewed real-world demands [1, 6, 18, 27], there is a high risk that a few nodes storing hot data may become hotspots while other nodes are relatively idle. In this paper, we argue that rigid load balancing schemes, i.e., passive recovery after timeout, is a major cause for long latency tails in coded storage, especially in the presence of skewed demands. In this case, we can actually reduce latency by proactively launching degraded reads for some requests to shift loads away from hotspots early.

Recently, there have been other studies to reduce download latency from coded storage systems, mainly leveraging redundant downloads [5, 10, 12, 22–24]. The idea is to download more than k coded blocks in a (k, r) RS-coded system to exploit a queueing-theoretical gain: as soon as the first k blocks are obtained, the remaining downloads can be stopped. However, such a scheme mainly benefits *non-systematic* codes, where there is no original copies of objects in each coding group. A latency optimization model has been proposed in [28] to jointly perform erasure code selection, content placement, and scheduling policy optimization, also for *non-systematic* codes. In contrast, we focus on systematic codes (where for each object, there is a single node storing its original copy) that are commonplace in production environments to allow normal reads. With systematic codes, always downloading k or more blocks is not efficient. Besides, we mainly focus on disk I/O bottlenecks due to queued read tasks instead of variable network latencies of downloads.

The power-of-two-choices algorithm [13, 14] is a classical randomized load balancing scheme with theoretical guarantees and wide applications [19]. Recently, there have been renewed interests to generate power-of-two load balancing to low-latency scheduling of batched tasks. Sparrow [15] proposes to schedule a batch of m tasks in a Spark job to multiple workers by selecting the lead loaded m out of dm probed workers. Later, it is theoretically shown [30] that a similar batch sampling technique maintains the same asymptotic performance as the power-of-two-choices algorithm while reducing the number of probes. Our per-request optimal decisions generalize the power-of-two idea to the load balancing between the normal read and different degraded reads in a erasure coded storage system, where the objective for comparison is not so obvious as in prior job scheduling literature. We propose the least-marginal-load-first policy can judiciously trades between current request latency and overall system efficiency. Moreover, we unify the proposed schemes in an optimization framework that can be executed lazily to further save probing overhead.

Many prior efforts have been devoted to reducing tail latencies in replication-based systems. Specifically, the *C*3 system in [7] presents a distributed approach to reducing the tail latency, stabilizing the behavior via a server ranking function that considers concurrent requests on the fly and penalizes those servers with long queue sizes. Tiny-Tail Flash [29] eliminates tail latencies induced by garbage collection by circumventing GC-blocked I/Os with four novel strategies proposed. One of such strategies is to proactively generate content of read I/Os that are blocked by ongoing GCs. In this work, we focus on reducing tail latencies in storage systems that are based on systematic erasure codes, by leveraging proactively launched degraded reads. Furthermore, we address the request concurrency issue in a much more complex situation with theoretically inspired methods.

6 CONCLUSION

Erasure-coding-based storage systems often suffer from long access latency tails. Prior studies have attributed this to the presence of degraded reads when the original data is unavailable and mainly aimed at improving coding structures to reduce degraded read costs. We take a radically different approach to tail latency reduction in coded storage systems and launch degraded reads intentionally and judiciously to balance the loads. Specifically, we propose a variety of schemes to direct loads based on either per-request decisions made from instantaneously probed storage node queue status or an optimized load direction table computed by statistical optimization with lazy queue status probes. We implemented a prototype system and deployed it on a cluster of 98 machines to evaluate the performance based on a large amount of real-world traces. Results suggest that the proposed least-marginal-load-first policy based on instantaneous sampled queue status can reduce the median request latency by more than 40% and the 95-percentile tail latency by more than 75% in both RS-coded systems and LRC-based systems, as compared to the existing approach of normal reads followed by passive degraded reads upon timeouts. The statistical optimization approach with lazy queue probing can also significantly reduce request and task latencies with a much lower system probing overhead.

REFERENCES

[1] Cristina L Abad, Nick Roberts, Yi Lu, and Roy H Campbell. 2012. A storage-centric analysis of mapreduce workloads: File popularity, temporal locality and arrival patterns. In *Workload Characterization (IISWC), 2012 IEEE International Symposium on*. IEEE, 100–109.

[2] MOSEK ApS. 2017. *The MOSEK Python optimizer API manual Version 7.1 (Revision 62)*. http://docs.mosek.com/7.1/pythonapi/index.html

[3] Dhruba Borthakur. 2008. HDFS architecture guide. *HADOOP APACHE PROJECT http://hadoop. apache. org/common/docs/current/hdfs design. pdf* (2008).

[4] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, et al. 2011. Windows Azure Storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 143–157.

[5] Shengbo Chen, Yin Sun, Longbo Huang, Prasun Sinha, Guanfeng Liang, Xin Liu, Ness B Shroff, et al. 2014. When queueing meets coding: Optimal-latency data retrieving scheme in storage clouds. In *IEEE INFOCOM 2014-IEEE Conference on Computer Communications*. IEEE, 1042–1050.

[6] Yanpei Chen, Sara Alspaugh, and Randy Katz. 2012. Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads. *Proceedings of the VLDB Endowment* 5, 12 (2012), 1802–1813.

[7] Aditya Ganjam, Faisal Siddiqui, Jibin Zhan, Xi Liu, Ion Stoica, Junchen Jiang, Vyas Sekar, and Hui Zhang. 2015. C3: Internet-Scale Control Plane for Video Quality Optimization. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, Oakland, CA, 131–144. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/ganjam

[8] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google file system. In *ACM SIGOPS operating systems review*, Vol. 37. ACM, 29–43.

[9] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, Sergey Yekhanin, et al. 2012. Erasure Coding in Windows Azure Storage.. In *Usenix annual technical conference*. Boston, MA, 15–26.

[10] Gauri Joshi, Emina Soljanin, and Gregory Wornell. 2015. Efficient replication of queued tasks to reduce latency in cloud systems. In *53rd Annual Allerton Conference on Communication, Control, and Computing*.

[11] Osama Khan, Randal C Burns, James S Plank, William Pierce, and Cheng Huang. 2012. Rethinking erasure codes for cloud file systems: minimizing I/O for recovery and degraded reads.. In *FAST*. 20.

[12] Guozheng Liang and Ulas C Kozat. 2014. Fast cloud: Pushing the envelope on delay performance of cloud storage with coding. *Networking, IEEE/ACM Transactions on* 22, 6 (2014), 2012–2025.

[13] Michael Mitzenmacher. 2001. The power of two choices in randomized load balancing. *Parallel and Distributed Systems, IEEE Transactions on* 12, 10 (2001), 1094–1104.

[14] Michael David Mitzenmacher. 1996. *The Power of Two Choices in Randomized Load Balancing*. Ph.D. Dissertation. UNIVERSITY of CALIFORNIA at BERKELEY.

[15] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. 2013. Sparrow: distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 69–84.

[16] KV Rashmi, Nihar B Shah, Dikang Gu, Hairong Kuang, Dhruba Borthakur, and Kannan Ramchandran. 2013. A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the Facebook warehouse cluster. *Proc. USENIX HotStorage* (2013).

[17] KV Rashmi, Nihar B Shah, Dikang Gu, Hairong Kuang, Dhruba Borthakur, and Kannan Ramchandran. 2014. A hitchhiker’s guide to fast and efficient data reconstruction in erasure-coded data centers. In *Proceedings of the 2014 ACM conference on SIGCOMM*. ACM, 331–342.

[18] Kai Ren, YongChul Kwon, Magdalena Balazinska, and Bill Howe. 2013. Hadoop’s adolescence: an analysis of Hadoop usage in scientific workloads. *Proceedings of the VLDB Endowment* 6, 10 (2013), 853–864.

[19] Andrea W Richa, M Mitzenmacher, and R Sitaraman. 2001. The power of two random choices: A survey of techniques and results. *Combinatorial Optimization* 9 (2001), 255–304.

[20] Maheswaran Sathiamoorthy, Megasthenis Asteris, Dimitris Papailiopoulos, Alexandros G Dimakis, Ramkumar Vadali, Scott Chen, and Dhruba Borthakur. 2013. Xoring elephants: Novel erasure codes for big data. In *Proceedings of the VLDB Endowment*, Vol. 6. VLDB Endowment, 325–336.

[21] Eric Schurman and Jake Brutlag. 2009. The user and business impact of server delays, additional bytes, and HTTP chunking in web search. In *Velocity Web Performance and Operations Conference*.

[22] Nihar B Shah, Kangwook Lee, and Kannan Ramchandran. 2014. The MDS queue: Analysing the latency performance of erasure codes. In *2014 IEEE International Symposium on Information Theory*. IEEE, 861–865.

[23] Nihar B Shah, Kangwook Lee, and Kannan Ramchandran. 2016. When do redundant requests reduce latency? *IEEE Transactions on Communications* 64, 2 (2016), 715–722.

[24] Yin Sun, Zizhan Zheng, C Emre Koksal, Kyu-Han Kim, and Ness B Shroff. 2015. Provably delay efficient data retrieving in storage clouds. *arXiv preprint arXiv:1501.01661* (2015).

[25] Itzhak Tamo and Alexander Barg. 2014. A family of optimal locally recoverable codes. *Information Theory, IEEE Transactions on* 60, 8 (2014), 4661–4676.

[26] Hakim Weatherspoon and John D Kubiatowicz. 2002. Erasure coding vs. replication: A quantitative comparison. In *Peer-to-Peer Systems*. Springer, 328–337.

[27] Mingyuan Xia, Mohit Saxena, Mario Blaum, and David A Pease. 2015. A tale of two erasure codes in HDFS. In *To appear in Proceedings of 13th Usenix Conference on File and Storage Technologies*.

[28] Yu Xiang, Tian Lan, Vaneet Aggarwal, and Yih Farn R Chen. 2014. Joint latency and cost optimization for erasurecoded data center storage. *ACM SIGMETRICS Performance Evaluation Review* 42, 2 (2014), 3–14.

[29] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A. Chien, and Haryadi S. Gunawi. 2017. Tiny-Tail Flash: Near-Perfect Elimination of Garbage Collection Tail Latencies in NAND SSDs. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*. USENIX Association, Santa Clara, CA, 15–28. https://www.usenix.org/conference/fast17/technical-sessions/presentation/yan

[30] Lei Ying, R Srikant, and Xiaohan Kang. 2015. The power of slightly more than one sample in randomized load balancing. In *2015 IEEE Conference on Computer Communications (INFOCOM)*. IEEE, 1131–1139.

[31] Yujia Zhu, James Lin, Patrick PC Lee, and Yan Xu. [n. d.]. Boosting Degraded Reads in Heterogeneous Erasure-Coded Storage Systems. ([n. d.]).