Facilitating Magnetic Recording Technology Scaling for Data Center Hard Disk Drives through Filesystem-level Transparent Local Erasure Coding

Yin Li[⋆]*, Hao Wang*[⋆]*, Xuebin Zhang*[†]*, Ning Zheng*[∗]*, Shafa Dahandeh*[†]*, and Tong Zhang*[⋆]
[⋆]*ECSE Department, Rensselaer Polytechnic Institute, USA*
[†] *EMC/DSSD, USA* [∗] *ScaleFlux, USA* [†] *Western Digital (WD), USA*

Abstract

This paper presents a simple yet effective design solution to facilitate technology scaling for hard disk drives (HDDs) being deployed in data centers. Emerging magnetic recording technologies improve storage areal density mainly through reducing the track pitch, which however makes HDDs subject to higher read retry rates. More frequent HDD read retries could cause intolerable tail latency for large-scale systems such as data centers. To reduce the occurrence of costly read retry, one intuitive solution is to apply erasure coding locally on each HDD or JBOD (just a bunch of disks). To be practically viable, local erasure coding must have very low coding redundancy, which demands very long codeword length (e.g., one codeword spans hundreds of 4kB sectors) and hence large file size. This makes local erasure coding mainly suitable for data center applications. This paper contends that local erasure coding should be implemented transparently within filesystems, and accordingly presents a basic design framework and elaborates on important design issues. Meanwhile, this paper derives the mathematical formulations for estimating its effect on reducing HDD read tail latency. Using Reed-Solomon (RS) based erasure codes as test vehicles, we carried out detailed analysis and experiments to evaluate its implementation feasibility and effectiveness. We integrated the developed design solution into *ext4* to further demonstrate its feasibility and quantitatively measure its impact on average speed performance of various big data benchmarks.

1 Introduction

Flash-based solid-state data storage has been rapidly displacing magnetic recording hard disk drives (HDDs) in traditional market segments, e.g., consumer and personal computing, and tier-1 (and even tier-2) enterprise storage. Although this trend will inevitably continue in the advent of 3D NAND flash memory, it by no means proclaims a doomed future for HDD technology. The blossoming data center and cloud-based infrastructure provide a new area with tremendous growth potential for HDDs. Nevertheless, as pointed out by a well-received position paper by Google Research [6], such a paradigm shift demands fundamental re-thinking on the design of HDDs and system architecture. Authors of [6] shared a collection view on open research opportunities and challenges for the new era of *data center HDDs*.

This paper studies the design of data center HDDs with the focus on leveraging workload characteristics to facilitate magnetic recording technology scaling. In particular, we are interested in how one could alleviate the HDD areal density vs. read retry rate conflict. As conventional magnetic recording technology approaches its limit at around 1Tb/in^2, the industry is now exploring a variety of new technologies including heat assisted magnetic recording (HAMR) [23,26], shingled magnetic recording (SMR) [15, 17], and two-dimensional magnetic recording (TDMR) [13,24,28]. All these new technologies improve bit areal density by shrinking the track pitch (i.e., the distance between adjacent tracks). However, due to the mechanical rotating nature of HDDs, a smaller track pitch inevitably makes HDDs more sensitive to head offset (i.e., the read/write head flies off the center of the target track). This leads to a higher probability of read retry. As a result, future magnetic recording technologies are increasingly subject to the conflict between areal density and read retry rate. The long latency penalty of HDD read retry may cause intolerable tail latency for data centers [6, 8].

Aiming to alleviate the areal density vs. read retry rate conflict for data center HDDs, this work is motivated by a very simple fact: if the data being stored in HDDs have inherent redundancy for error correction, we may recover the failed sectors through system-level error correction (at tens or hundreds of μs latency) other than HDD read retry (at tens or hundreds of ms latency). One may argue that RAID and distributed erasure coding already conveniently provide such a feature. However, RAID is being replaced by distributed erasure coding in data centers, and using distributed erasure coding to mitigate HDD read retry can cause significant overheads (e.g., network traffic), in particular under high HDD retry rates (e.g., 10^{-4} and above). This work focuses on the scenarios of storing user data and associated coding redundancy locally together in one HDD or JBOD (just a bunch of disks) and recovering failed sectors locally at the server that directly connects to the HDDs. It is referred to as *local erasure coding* in this work. We note that local erasure coding only aims to reduce the occurrence of costly HDD read retry operations and does not provide any guarantee for tolerating catastrophic HDD failures. Hence, it is completely orthogonal to distributed erasure coding, and must have very small coding redundancy.

A *soft sector read failure* occurs when an HDD fails to decode one sector during its normal operation. Let p_h denote the soft sector read failure probability, which is HDD read retry rate in current practice, i.e., the probability that an HDD switches from the normal operation mode into a retry mode to repeatedly read the failed sector by adjusting the read head position. Current HDDs keep p_h relatively low (e.g., 10^{-6} and below). The probability that one sector cannot be correctly read even after the long-latency HDD read retry is called hard sector failure rate, which must be extremely low (e.g., 10^{-14} and below). When using the local erasure coding, HDDs do not immediately switch to the retry mode upon a soft sector read failure, instead we first try to fix it through the local erasure decoding. Only when the erasure decoding fails, HDDs switch into the read retry mode. Let p_s denote the probability that the local erasure decoding fails. We should minimize the coding redundancy and meanwhile ensure $p_s << p_h$. In order to minimize the local erasure coding redundancy, we must use long codeword length. Therefore, local erasure coding should be applied to systems with dominantly large flies, e.g., data centers.

In spite of the simple basic idea, its practical realization is non-trivial. The first question is whether the local erasure coding should be implemented at the application layer, OS layer, or inside HDD. As elaborated later in Section 2.2, we believe that it should be implemented by the local filesystem at the OS layer with complete transparency to the upper application layer. Integrating local erasure coding into filesystem is far beyond merely implementing a high-speed encoding/decoding library, and one has to modify and enhance the filesystem architecture. This work uses the journaling filesystem *ext4* as a test vehicle to study the integration of local erasure coding into the filesystem. In particular, we investigate the separate treatment of data and filesystem metadata, and develop techniques to handle scenarios when the HDD write is unaligned with the erasure codeword boundary and when fine-grained data update occurs. Meanwhile, we derive mathematical formulations for estimating its effectiveness on reducing HDD read tail latency.

We carried out a variety of analysis and experiments to study the effectiveness and feasibility of the proposed design solution, where we use Reed-Solomon (RS) codes as local erasure codes. Our analysis results show that RS codes with the coding redundancy of less than 2% can reduce the 99-percentile latency by more than 65% when p_h is 1×10^{-3}. Since the local erasure encoding and decoding can add noticeable extra latency into the data I/O path, it will degrade the average system speed performance compared with the ideal retry-free scenario. To evaluate such impact and meanwhile further demonstrate the practical feasibility of filesystem-level transparent local erasure coding, we integrate the proposed design solution into Linux kernel 3.10.102 I/O path (in particular the VFS and *ext4*), and carried out experiments using the big data benchmark suite HiBench 3.0 [1]. Motivated by the emergence of data center CPUs with built-in FPGA (e.g., the Xeon CPU with built-in FPGA as lately announced by Intel), we investigated the use of both software-based and hardware-based RS coding engine. The measurement results show that, even under p_h of 10^{-3}, local erasure coding only incurs (much) less than 3.5% average speed performance degradation. Finally, we also present results on the storage capacity overhead under various benchmarks in HiBench 3.0, and the latency overhead induced by fine-grained update.

2 Background and Rationale

2.1 Magnetic Recording Technologies

To move the storage areal density beyond 1Tb/in^2 and towards 10Tb/in^2 over the next decade, the HDD industry is exploring several new technologies including HAMR, SMR, and TDMR, all of which improve areal density through significantly reducing the track pitch. A smaller track pitch results in a stronger inter-track interference (ITI) and hence worse signal-to-noise ratio, which makes HDDs more sensitive to read head off-set. This can be illustrated in Fig. 1. The read head off-set is defined as the distance between the target track center and head center. The perfect head-track alignment (i.e., zero read head off-set) corresponds to the minimal ITI and hence best read channel signal processing performance, leading to the minimal sector read failure probability. Nevertheless, the mechanical disk rotation inevitably causes run-time fluctuation of the read head position. As shown in Fig. 1, the same read head off-set induces stronger ITI from neighboring track (i.e., track N-1 in the figure) in HDDs with a smaller track pitch. A stronger ITI directly results in worse read channel signal processing performance and hence a higher sector read failure probability. Therefore, regardless to the specific magnetic recording technology, HDDs are fundamentally subject to areal density vs. read retry rate conflict.

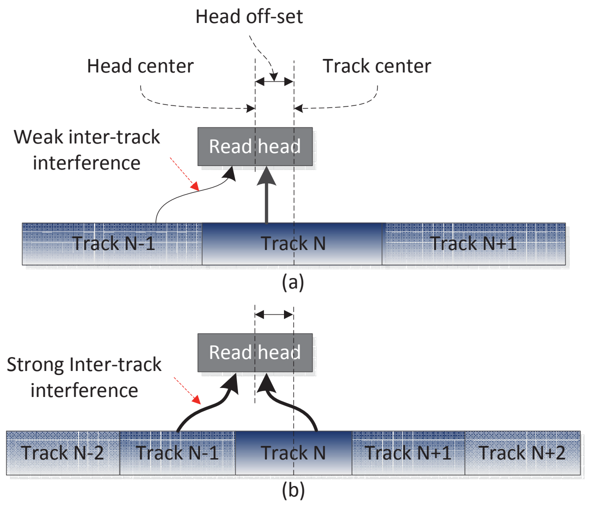


Figure 1: Illustration of different effect of head off-set under (a) large track pitch and (b) small track pitch, where the track-N is the target track being read.

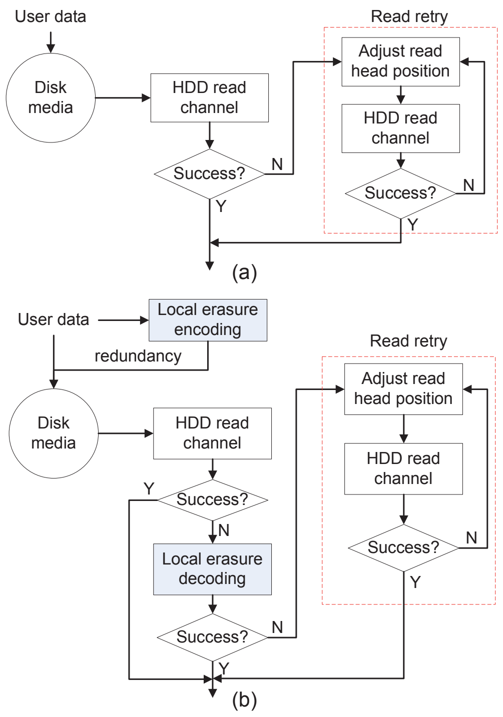


Figure 2: Illustration of (a) current practice handling sector read failures, and (b) using the local erasure coding to reduce the occurrence of read retry.

2.2 Local Erasure Coding

Local erasure coding aims to reduce the occurrence of HDD read retry by storing additional coding redundancy together with the original user data in the same HDD or JBOD. Fig. 2 illustrates the simple basic concept: In current practice, once the read channel signal processing fails during the normal operation (most likely due to runtime read head off-set), HDDs switch into the read retry mode to repeatedly read the failed sector by adjusting the read head position. When using local erasure coding, the user data and associated coding redundancy are stored together locally. Upon a soft sector read failure, we first try to recover the data through the local erasure decoding, and invoke HDD read retry only if the local erasure decoding fails, as shown in Fig. 2(b).

Along the data storage hierarchy, we could implement the local erasure coding at either application layer, OS layer, or hardware layer inside HDDs. With the full knowledge about their own data access characteristics, applications can best optimize the use of local erasure coding. Nevertheless, the efforts of integrating/optimizing the local erasure coding in each application may not be justified in practice. Moreover, not all the data being stored on HDDs are visible to applications (e.g., filesystem metadata). On the other hand, although intra-HDD implementation keeps the software stack completely intact, it has several problems: (1) Because filesystem metadata have fine-grained access characteristics, we should configure the local erasure coding differently for metadata and data. However, HDD on its own cannot distinguish between data and metadata. (2) HDDs are not aware of the validity of data in each sector (i.e., whether or not the data are being used by the filesystem), which could cause a large number of unnecessary local erasure encoding operations and disk rotations, especially under long codeword length. (3) Local erasure coding could leverage data cached in memory to reduce the disk access. HDDs typically have very small internal cache memory, especially compared with the host DRAM. As a result, intra-HDD realization of local erasure coding is subject to more disk rotations.

In comparison, OS (in particular filesystem) can the most appropriate place to implement the local erasure coding. On one hand, filesystem-based implementation is completely transparent to the upper application layers, which lowers the barrier of deploying the local erasure coding in real systems. On the other hand, with the full awareness and control of data access/storage on HDDs, the filesystem can effectively optimize the realization of local erasure coding. Therefore, this work focuses on filesystem-level transparent local erasure coding.

2.3 Choosing the Code

An error correction code (ECC) is characterized by four parameters $\{k,m,w,d\}$, i.e., each codeword protects k user data symbols using m redundant symbols (hence the codeword length is $k+m$), each symbol contains w bits, and the minimum codeword distance is d. For any linear ECC code, its minimum distance d is subject to the well-known Singleton bound, i.e., $d \leq m+1$ [16]. A code that achieves the equality in the Singleton bound is called an MDS (maximum distance separable) code, which can achieve the maximum guaranteed error and erasure correction strength. As the most well-known MDS code, RS code [27] has been used in numerous data communication and storage systems. An ECC with the minimum distance of d can correct up to $\lfloor \frac{d-1}{2} \rfloor$ errors or up to $d-1$ erasures (note that the term *erasure* means an error with the known location). Hence, when being used for erasure coding (i.e., the location of all the errors is known prior decoding), a (k,m) RS code guarantees to correctly recover up to m erasures within one codeword.

This work uses RS codes to realize the local erasure coding. RS codes are typically constructed over binary Galois Field (GF). Given the underlying GF with the order of 2^w, the codeword length can be up to 2^w-1 and 2^w for cyclic and non-cyclic RS codes, respectively, and each symbol contains w bits. Non-cyclic RS codes (e.g., the widely used Cauchy RS codes [5, 20]) are primarily used for erasure coding, and their encoding/decoding are realized through GF matrix-vector multiplication (and GF matrix inversion for decoding). Cyclic RS codes have a much richer set of encoding/decoding algorithms and can more conveniently handle both errors and erasures. Interested readers are referred to [16, 27] for details.

3 Proposed Design Solution

This section first presents the basic framework on realizing the filesystem-level transparent local erasure coding, then mathematically formulates its effect on reducing the read tail latency, and finally presents solutions to address two non-trivial issues for its practical implementation.

3.1 Basic Framework

When implementing transparent local erasure coding, it is important to treat filesystem metadata and user data differently. Leveraging the large file size and typically coarse-grained data access patterns in data centers, we apply long RS codes to user data on the per-file basis, i.e., each RS codeword spans over hundreds of 4kB sectors and all the data within one RS codeword belong to the same file. This is illustrated in Fig. 3: One (k,m,w) RS codeword spans over $k+m$ consecutive sectors and each w-bit symbol comes from one sector. Hence, HDD read failures on any m sectors could be recovered by RS code decoding. To simplify the implementation, we use the same (k,m,w) RS code for all the files. Let N denote the number of 4kB sectors in one file. The filesystem partitions all the N sectors into $\lceil \frac{N}{k} \rceil$ groups, and appends m sectors to each group for the storage of the RS coding redundancy. The last group may have $k' < k$ sectors, for which we use a shortened (k',m,w) RS code. Note that the shortened (k',m,w) RS code shares the same encoder and decoder as the original (k,m,w) RS code by simply setting the content of the other $k-k'$ sectors as zeros.

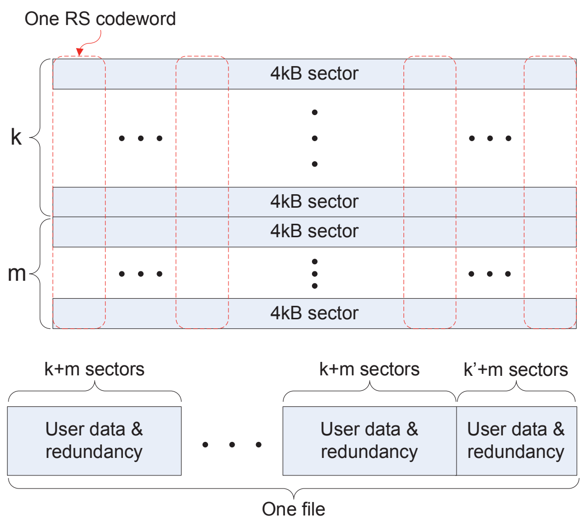


Figure 3: Illustration of per-file local erasure coding.

Let p_h denote the HDD soft sector read failure probability, and let $p_s << p_h$ denote the target local erasure decoding failure probability (i.e., reduce the HDD read retry rate from p_h to p_s via the local erasure coding). Given the value of k, we should search for the minimum value of m subject to

$$\sum_{i=m+1}^{k+m} \binom{k+m}{i} p_h^i \cdot (1-p_h)^{(k+m-i)} \leq p_s. \qquad (1)$$

To illustrate the dependence of coding redundancy ratio (defined as m/k) on the codeword length, we set p_s as 10^{-8} and calculate the required coding redundancy over different k and p_h, as shown in Fig. 4. The results show that, in order to minimize the coding redundancy, we must deploy long RS codes.

When the local erasure coding is implemented on the per-file basis, the average storage capacity overhead can be calculated as follows. Let $g(x)$ denote the probability density function (PDF) of the file size, where x denotes size of one file in terms of the number of 4kB sectors. We

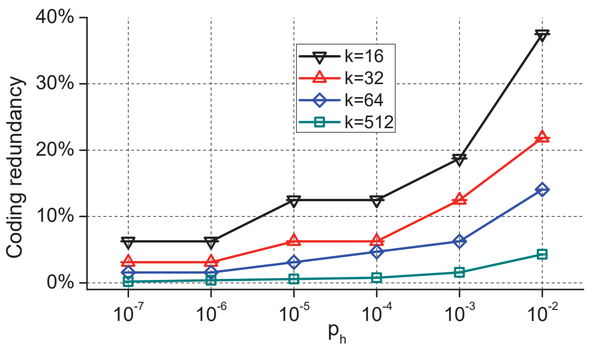


Figure 4: Calculated coding redundancy under different values of k and p_h, where k is the number of user data symbols per codeword and p_h is the soft sector read failure probability. The target decoding failure rate is 10^{-8}.

can express the average storage capacity overhead r_{avg} as

$$r_{avg} = \frac{\int_0^\infty g(x) \cdot \lceil \frac{x}{k} \rceil \cdot m \, dx}{\int_0^\infty g(x) \cdot x \, dx}. \quad (2)$$

This clearly shows that the space overhead reduces as the file size increases, and the minimum overhead per file is m sectors no matter how small the file is. For large files, the overhead can be approximated as m/k (i.e., m redundant sectors per k user data sectors), e.g., the overhead is about 0.39% when using a (1019, 4) RS code.

To accommodate the fine access granularity of filesystem metadata, we apply erasure coding to the filesystem metadata on the per-sector basis, i.e., erasure coding reduces to data replication. In particular, to store one sector of filesystem metadata, we allocate m' consecutive sectors to store m' replicas of the same sector content. Given the soft sector failure probability p_h and the target local erasure decoding failure probability $p_s << p_h$, we should search for the minimum value of m' subject to

$$p_h^{m'} \leq p_s. \quad (3)$$

Since p_h should not be too high (e.g., 10^{-2} and below), a small value of m' (e.g., 3 or 4) is sufficient to make p_s small enough. Therefore, the m' consecutive sectors most likely reside on the same track, and it does not incur noticeable HDD access latency overhead. Among all the m' replica sectors, we designate the first sector as the lead sector and the following $m'-1$ sectors as shadow sectors. All the pointers in the filesystem metadata structure point to the lead sector, and all the shadow sectors are only used for tolerating HDD sector read failures. Using *ext4* as an example, Fig. 5 illustrates its use in the context of inode pointer structure. Each sector storing inode or singly/doubly/triply indirect blocks is replicated m' times, and the pointed data sectors fall into one or multiple local erasure coding groups within the same file.

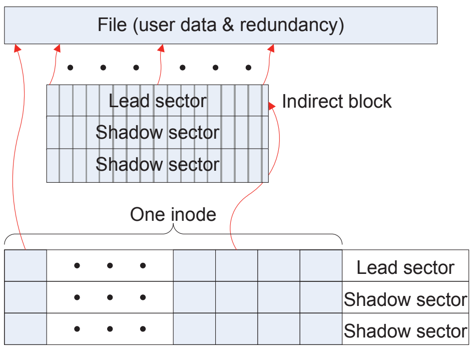


Figure 5: Illustration of replica-based protection for inode.

In order to practically implement this design strategy, the host-to-HDD interface protocol and HDD firmware should be appropriately modified. In particular, the host should be able to notify the HDD whether the host would like the HDD to simply return an immediate error message, upon a read failure, without internal retry. The HDD firmware should support two operational modes: skip a failed sector without retry, or immediately invoke read retry upon a read failure.

Although the above basic framework is indeed very simple and straightforward, its practical implementation involves the following two issues:

- *Unaligned HDD write*: Unaligned HDD write occurs when the user data being written to the HDD do not exactly fill one or multiple coding groups. Ideally we hope to align HDD write with the local erasure coding group boundary, i.e., we can always first accumulate each group of k consecutive user data pages in the host memory, then carry out the local erasure encoding and write the total $k+m$ pages to the HDD. Unfortunately, this may not be always possible in practice, e.g., in the case of direct I/O and synchronous I/O. Even in the case of asynchronous I/O, filesystem must carry out periodic flushing, which could make disk write unaligned with the local erasure coding group boundary.
- *Fine-grained data update*: Although our interested data center applications have predominantly coarse-grained data access, especially data write, it is not impossible to have fine-grained data update (e.g., update of tens of 4kB sectors) in practice. Given the very long RS codeword length, it is not trivial to effectively support such fine-grained data update.

In the remainder of this section, we first derive mathematical formulations for estimating the effectiveness of using local erasure coding to reduce HDD read tail latency, and then present techniques to address the above two non-trivial implementation issues.

3.2 Tail Latency Estimation

The objective of local erasure coding is to relax the constraint on HDD soft sector read failure probability without sacrificing HDD read tail latency. This subsection derives mathematical formulations for estimating read tail latency with and without using the local erasure coding. We consider the scenario of reading N consecutive sectors from an HDD, and let T denote the latency of successfully reading all the N sectors. Due to the varying additional latency caused by read retry and RS coding, T can be considered as a discrete variable. Therefore, in order to obtain the tail latency (e.g., 99-percentile latency), we should derive its discrete probability distribution, also known as probability mass function (PMF), denoted as $f(T)$. Given the target tail percentage P_{tail} (e.g., 99% or 99.9%), we can search for the tail latency T_{tail} subject to

$$\sum_{T=0}^{T_{tail}} f(T) \geq P_{tail}. \tag{4}$$

Let τ_{retry} denote the latency to recover one sector during the read retry mode, and τ_u denote the latency for an HDD to read one sector during its normal mode. Note that τ_{retry} is approximately the latency of one or more disk rotations. Since we are interested in comparing the latency with and without using the local erasure coding, we omit the HDD seek latency. Recall that p_h denote the soft sector read failure probability. Based on our interaction with HDD vendors, it is reasonable to approximately model the soft sector read failure as an independent and identical distributed (i.i.d.) random variable. Therefore, in this work, we assume there is no correlation in soft sector errors between contiguous sectors. Recall that $f(T)$ denote the probability mass function of the latency T. Hence, we have the formulations for the case without using the local erasure coding:

$$\begin{cases} T = t \cdot \tau_{retry} + \tau_u \cdot N \\ f(T) = \binom{N}{t} p_h^t \cdot (1-p_h)^{(N-t)} \end{cases} \tag{5}$$

where $t \geq 0$ is the number of soft sector read failures.

Next, let us consider the case of using the local erasure coding. Given the coding parameters k and m, we can calculate the local erasure decoding failure probability p_s using Eq. (1). Define $l = \lfloor N/k \rfloor$, and $k' = N - l \cdot k$, i.e., there are total l complete $(k+m)$-sector groups followed by one shortened $(k'+m)$-sector group. Since local erasure decoding can be carried out concurrently with HDD sector read, we assume that only the decoding of the last group contributes additional latency to the overall latency. The formulations for T and $f(T)$ are derived for the following four different cases:

1. Case I: None of the first l groups suffers from local erasure decoding failure, and none of the last k' user data sectors suffers from soft sector read failure:

$$\begin{cases} T = \tau_u \cdot (N + l \cdot m) \\ f(T) = (1-p_s)^l \cdot (1-p_h)^{k'} \end{cases} \tag{6}$$

2. Case II: None of the first l groups experiences local erasure decoding failure, but $e_0 \leq m$ sector read failures occur in the last $(k'+m)$ sectors. Let $\tau_{dec}(e_0)$ denote the latency to correct e_0 sectors, we have

$$\begin{cases} T = \tau_u \cdot \big(N + (l+1) \cdot m\big) + \tau_{dec}(e_0) \\ f(T) = (1-p_s)^l \cdot \binom{k'+m}{e_0} \cdot p_h^{e_0} \\ \qquad \cdot (1-p_h)^{(k'+m-e_0)} \end{cases} \tag{7}$$

3. Case III: Among the first l groups, j groups experience local erasure decoding failures due to $e_1, e_2, \cdots, e_j > m$ sector read failures, and $e_0 \leq m$ soft sector read failures occur in the last $(k'+m)$-sector group. Let p_i denote the probability that e_i soft sector read failures occur within one group, where $p_i = \binom{k+m}{e_i} \cdot p_h^{e_i} \cdot (1-p_h)^{(k+m-e_i)}$, we have

$$\begin{cases} T = \tau_u \cdot \big(N + (l+1) \cdot m\big) + j \cdot \tau_{dec}(m) \\ \qquad + \tau_{dec}(e_0) + \sum_{i=1}^{j} \tau_{retry} \cdot (e_i - m) \\ f(T) = \Big(\frac{l!}{(l-j)!} \cdot \prod_{i=1}^{j} p_i\Big) \cdot (1-p_s)^{(l-j)} \\ \qquad \cdot \binom{k'+m}{e_0} p_h^{e_0} \cdot (1-p_h)^{(k'+m-e_0)} \end{cases} \tag{8}$$

4. Case IV: It only differs from the Case III in that the last $(k'+m)$-sector group suffers from local erasure decoding failure as well (i.e., $e_0 > m$), and we have

$$\begin{cases} T = \tau_u \cdot \big(N + (l+1) \cdot m\big) + (j+1) \cdot \\ \qquad \tau_{dec}(m) + \sum_{i=0}^{j} \tau_{retry} \cdot (e_i - m) \\ f(T) = \Big(\frac{l!}{(l-j)!} \cdot \prod_{i=0}^{j} p_i\Big) \cdot (1-p_s)^{(l-j)} \end{cases} \tag{9}$$

When using the Eq. (4) to estimate the tail latency, we should exhaustively exam all the four cases above to obtain a complete PMF $f(T)$ of the access latency.

3.3 Addressing Unaligned HDD Write

We can formulate an unaligned HDD write as follows: A vector $\mathbf{d} = [\mathbf{d}_1, \mathbf{d}_2]$ contains k user data symbols in one codeword, where sub-vectors $\mathbf{d}_1$ and $\mathbf{d}_2$ contain k_1 and k_2 user data symbols ($k_1 + k_2 = k$). Unaligned HDD write occurs once the filesystem must write data to HDD when only $\mathbf{d}_1$ is available. The straightforward solution is to store dynamically shortened RS codewords on HDDs, as illustrated in Fig. 6(a). This however suffers from two problems: (1) Filesystem must accordingly record the length of each shortened codeword in the file metadata, which could noticeably complicate filesystem design. (2) The average coding redundancy could increase and hence degrade the effective HDD bit cost.

Therefore, we should still use the same RS code with the fixed k in the presence of unaligned HDD write. To achieve this objective, we propose a cache-assisted progressive encoding strategy. Recall that ECC encoding can be modeled as a matrix-vector multiplication $\mathbf{r} = \mathbf{G} \cdot \mathbf{d}$, where $\mathbf{r}$ represents the m redundant symbols to be computed, $\mathbf{d}$ represents the k user data symbols, and the $m \times k$ matrix $\mathbf{G}$ is the generator matrix. We define two length-k vectors $\mathbf{d}^{(1)} = [\mathbf{d}_1, \mathbf{O}]$ and $\mathbf{d}^{(2)} = [\mathbf{O}, \mathbf{d}_2]$, where $\mathbf{O}$ represents an all-zero vector. Because the RS codes are constructed over binary GF, we have $\mathbf{d} = \mathbf{d}^{(1)} \oplus \mathbf{d}^{(2)}$, where $\oplus$ represents bit-wise XOR operation. Therefore, the encoding procedure can be written as

$$\mathbf{r} = \mathbf{G} \cdot \mathbf{d} = \mathbf{G} \cdot \mathbf{d}^{(\mathbf{1})} \oplus \mathbf{G} \cdot \mathbf{d}^{(\mathbf{2})}. \tag{10}$$

Define $\mathbf{r}^{(1)} = \mathbf{G} \cdot \mathbf{d}^{(\mathbf{1})}$ and $\mathbf{r}^{(2)} = \mathbf{G} \cdot \mathbf{d}^{(\mathbf{2})}$, we have that $\mathbf{r} = \mathbf{r}^{(1)} \oplus \mathbf{r}^{(2)}$. As illustrated in Fig. 6(b), the filesystem first writes the k_1 sectors and associated m sectors storing $\mathbf{r}^{(1)}$ to the HDD, and meanwhile keeps $\mathbf{r}^{(1)}$ in OS page cache. Once the subsequent k_2 sectors are ready to be written to the HDD, the filesystem carries out encoding to obtain $\mathbf{r}^{(2)}$ and then compute the overall redundancy $\mathbf{r} = \mathbf{r}^{(1)} \oplus \mathbf{r}^{(2)}$. Finally, filesystem appends k_2 sectors storing $\mathbf{d}_2$ and m sectors storing $\mathbf{r}$ after the previous k_1 sectors (i.e., overwrites the previously written m sectors storing $\mathbf{r}^{(1)}$). In this way, we can ensure that the filesystem always uses the same fixed-length RS codewords on each file (except the last portion of the file). Since m is typically very small (e.g., 5 or 10) and we only need to keep one intermediate coding redundancy (i.e., $\mathbf{r}^{(1)}$) for each file, this design strategy will not cause noticeable cache space overhead. Clearly, this design strategy can be applied recursively when one group of k sectors is written to the HDD in more than two batches.

Finally, we note that, if power failure occurs when we overwrite $\mathbf{r}^{(1)}$, the previously written data $\mathbf{d}_1$ are not protected by valid local erasure code and hence are more subject to read retry. This however will not cause any storage integrity degradation since we only use local erasure coding to mitigate soft sector read failure.

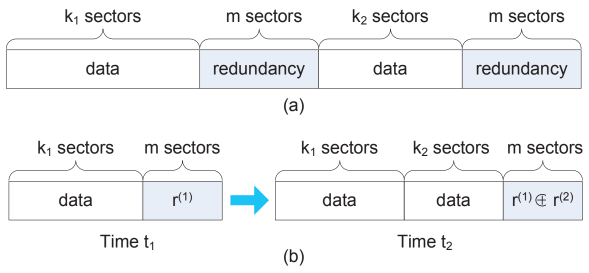


Figure 6: Addressing unaligned HDD write using (a) straightforward dynamic codeword shortening, and (b) proposed cached-assisted progressive encoding, where $k_1 + k_2 = k$.

3.4 Addressing Fine-grained Data Update

In the case of fine-grained data update, filesystem must carry out read-modify-write operations to update the m-sector coding redundancy. This may lead to noticeable system performance degradation, especially if the fine-grained data update is triggered by synchronous writes and meanwhile the other data in the same coding group are not present in host memory. We propose a two-phase write procedure to address this issue. As pointed out above, since local erasure coding only mitigates soft sector read failures, it does not degrade data storage integrity even if some data are temporarily not protected by local erasure code. In another word, when the m-sector coding redundancy becomes invalid due to fine-grained data update, the associated k sectors are simply subject to a higher risk of HDD read retry without any loss on their storage integrity. This observation directly suggests the use of a two-phase write procedure: Upon a fine-grained data update caused by synchronous writes, the filesystem first serves the synchronous write request as in current practice without modifying the m-sector coding redundancy. Then filesystem updates the corresponding m-sector coding redundancy asynchronously during background operations. Note that, if the fine-grained data update is not triggered by synchronous writes, the filesystem can merge these two phases together.

The next question is how to update the m-sector coding redundancy. To simplify the discussion, let us consider the scenario where we update the k user data symbols in one RS codeword on HDD from $\mathbf{d} = [\mathbf{d}_1, \mathbf{d}_2]$ to $\mathbf{d}' = [\mathbf{d}'_1, \mathbf{d}_2]$, where the unchanged content $\mathbf{d}_2$ is not cached in memory. To accordingly update the coding redundancy from $\mathbf{r} = \mathbf{G} \cdot \mathbf{d}$ to $\mathbf{r}' = \mathbf{G} \cdot \mathbf{d}'$, we have the following two different strategies:

1. We simply update the redundancy in the most straightforward manner, i.e., read the unchanged content $\mathbf{d}_2$ from HDD, then obtain the new redundancy $\mathbf{r}'$ by computing the complete matrix-vector

multiplication $\mathbf{G}\cdot\mathbf{d}'$, and finally write $\mathbf{r}'$ to HDD. Let k_1 denote the length of $\mathbf{d}_1$ (i.e., the data being updated). In order to update the m-sector coding redundancy, we need to read $k-k_1$ sectors from HDD.

2. The second option updates the redundancy through indirect computation. As discussed above in Section 3.3, we define $\mathbf{d}^{(1)} = [\mathbf{d}_1, \mathbf{O}]$, $\mathbf{d}^{(1)'} = [\mathbf{d}_1', \mathbf{O}]$, and $\mathbf{d}^{(2)} = [\mathbf{O}, \mathbf{d}_2]$. We define $\mathbf{r}^{(1)} = \mathbf{G}\cdot\mathbf{d}^{(1)}$, $\mathbf{r}^{(1)'} = \mathbf{G}\cdot\mathbf{d}^{(1)'}$, and $\mathbf{r}^{(2)} = \mathbf{G}\cdot\mathbf{d}^{(2)}$. The original redundancy can be expressed as $\mathbf{r} = \mathbf{r}^{(1)} \oplus \mathbf{r}^{(2)}$, which can also be re-written as $\mathbf{r} \oplus \mathbf{r}^{(1)} = \mathbf{r}^{(2)}$. As a result, we can express the updated redundancy $\mathbf{r}'$ as

$$\mathbf{r}' = \mathbf{r}^{(1)'} \oplus \mathbf{r}^{(2)} = \mathbf{r}^{(1)'} \oplus \mathbf{r} \oplus \mathbf{r}^{(1)}. \qquad (11)$$

Therefore, we read the original content $\mathbf{d}_1$ and original redundancy $\mathbf{r}$ from HDD, and accordingly generate the updated redundancy $\mathbf{r}'$ based upon Eq. (11). Using this strategy, in order to update the m-sector coding redundancy, we need to read k_1+m sectors from HDD.

These two strategies can be directly generalized for more complicated fine-grained data update (i.e., multiple separate regions within the same k-sector group are updated).

4 Analysis and Experiments

We evaluate the proposed design solution mainly from two aspects: (1) Improvement on tail latency: This proposed design solution aims to mitigate the impact of HDD technology scaling on tail latency. Using the mathematical formulations presented in Section 3.2, we show analysis results of HDD tail latency with and without using the proposed design solution under different configurations. (2) Impact on average system speed performance: The on-the-fly local erasure encoding and decoding can add noticeable extra latency into the data I/O path, which could degrade the average system speed performance. We use a variety of benchmarks in big data benchmark suite HiBench 3.0 [1] to evaluate the impact on average system speed performance. Moreover, we will present results on the storage capacity overhead under HiBench 3.0 benchmarks, and the latency overhead induced by fine-grained update.

4.1 Implementation of Coding Engine

We first discuss the construction of the RS codes being used in this study and their encoder/decoder implementation. We set the target local erasure code decoding failure probability p_s as 10^{-6}. Recall that p_h denotes the soft sector read failure probability. We consider four different values of p_h, including 1×10^{-4}, 5×10^{-4}, 1×10^{-3}, and 5×10^{-3}, and set the target codeword length as 255 and 1023. Accordingly, we can calculate the code parameters k and m, which are listed in Table 1. Note that the number of bits per symbol (i.e., w) is 8 and 16 when the codeword length is 255 and 1023, respectively.

Table 1: Parameters of RS-based local erasure codes.

p_h	$m+k=$255		$m+k=$1023	
	m	k	m	k
1×10^{-4}	3	252	4	1019
5×10^{-4}	4	251	7	1016
1×10^{-3}	5	250	9	1014
5×10^{-3}	9	246	19	1004

Based upon the open-source library jerasure [2], we developed and integrated an RS coding library into *ext4* in Linux kernel 3.10.102. Both encoding and decoding are realized through direct matrix-based computation instead of polynomial-based computation, which can readily leverage the on-chip cache resource in CPUs to maximize the throughput. We measured its encoding and decoding throughput on a PC with a 3.30GHz CPU and 8GB DRAM, and the results are shown in Fig. 7.

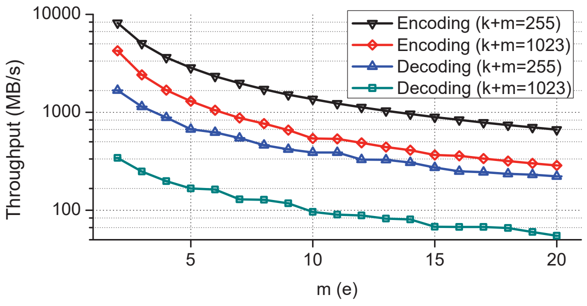
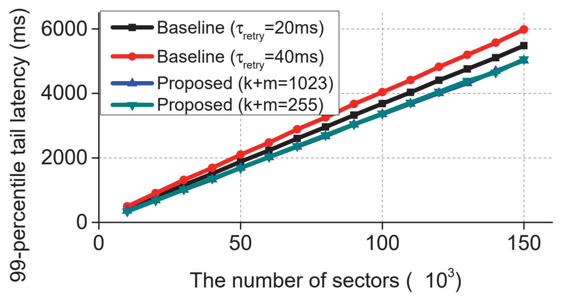


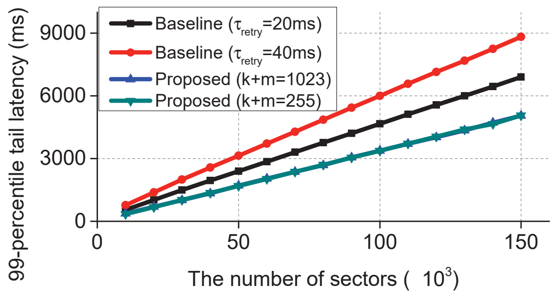
Figure 7: Measured encoding and decoding throughput under different value of m or the number of erasures e.

Let e denote the number of erased symbols per codeword. Given the same codeword length, the encoding (decoding) throughput reduces as m (e) increases. This is because the size of the matrices involved in encoding (decoding) is proportional the value of m (e). The results show that decoding throughput is significantly lower than encoding throughput under the same m and e. For example, with the codeword length of 255, the encoding throughput is about 1.5GB/s at $m=10$ while the decoding throughput is only 400MB/s at $e=10$. Fortunately, the probability that one codeword contains e erased symbols exponentially reduces as e increases. For example, with the codeword length of 255 and p_h of 1×10^{-3}, the probability that one codeword contains 2, 4, and 6 erased symbols is 2.5×10^{-2}, 1.3×10^{-4}, and 2.8×10^{-7}.

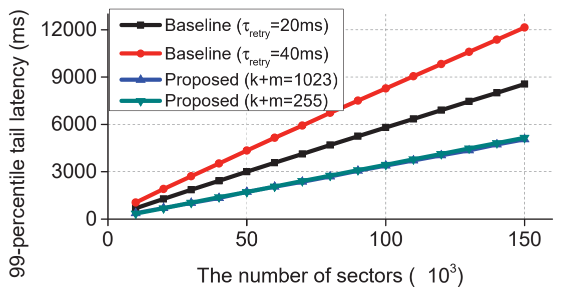
In addition, motivated by the emerging trend of integrating CPU with FPGA in one chip package (e.g., the



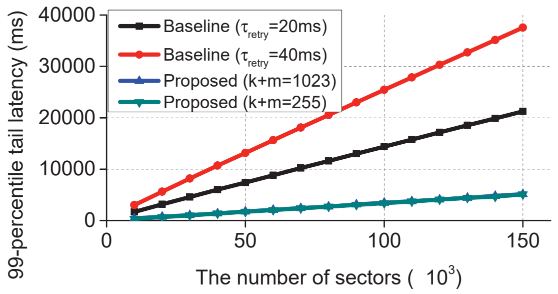
(a)



(b)



(c)



(d)

Figure 8: Calculated 99-percentile latency when reading different number of consecutive sectors with (a) $p_h = 10^{-4}$, (b) $p_h = 5 \times 10^{-4}$, (c) $p_h = 10^{-3}$, and (d) $p_h = 5 \times 10^{-3}$.

Xeon processors with built-in FPGAs announced by Intel at Open Compute Project (OCP) Summit 2016), we also studied the hardware-based implementation of RS coding engine. With abundant hardware-level parallelism and very high-speed CPU-FPGA interconnect, offloading RS encoding and decoding into the built-in hardware accelerator can significantly improve the achievable operational throughput. In this work, we designed parallel polynomial-based RS encoder, and RS decoder using the well-known Berlekamp-Massey algorithm and the parallel architecture presented in [14]. The design was carried out at the RTL level and Table 2 lists the synthesis results (in terms of equivalent XOR gate count) for all the eight RS codes, where all the implementations have the same throughput of 4GB/s with the clock frequency of 250MHz. The results show that, for the same RS code, the decoder consumes about 10x more silicon resource than the encoder at the same 4GB/s throughput. The decoder gate counts range from 156k to 894k, which can readily fit into modern FPGA devices.

Table 2: Hardware-based RS encoder/decoder implementation synthesis results.

Code Parameters		Equivalent XOR Gate Count	
m	k	Encoder	Decoder
3	252	11k	156k
4	251	11k	161k
5	250	17k	185k
9	246	28k	232k
4	1019	16k	634k
7	1016	31k	699k
9	1014	39k	732k
19	1004	78k	894k

4.2 HDD Tail Latency

Applying the mathematical formulations presented in Section 3.2, we computed the 99-percentile tail latency when reading consecutive N sectors from HDD, where N ranges from 10k to 150k (i.e., the data volume ranges from 40MB to 600MB). Recall that τ_{retry} represents the latency to recover one sector during the read retry mode, and τ_u represents the latency for HDD to read one sector during its normal mode. Assuming the use of 7200rpm HDD, we set τ_u as 33μs. Since read retry latency could significantly vary in practice, we treat τ_{retry} as the average read retry latency. Based upon our communications with HDD vendors, we consider two different values of τ_{retry}: 20ms and 40ms. The results are shown in Fig. 8.

The results show the effectiveness of using the local erasure coding to reduce the HDD read tail latency in the presence of high soft sector read failure probabilities. Since we constructed the RS codes with the target decoding failure probability of 10^{-6}, different value of per-

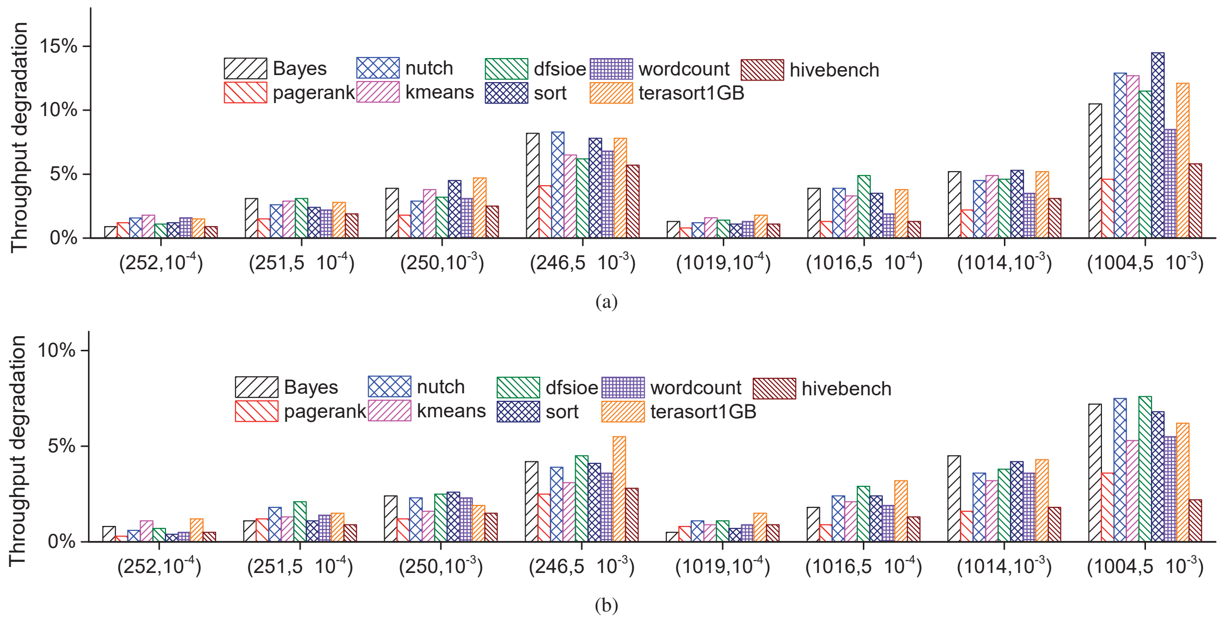


Figure 9: Measured throughput degradation of all the benchmarks when using (a) software-based and (b) hardware-based RS coding engine implementation. Each code is represented as (k, p_h).

sector read retry latency τ_{retry} (i.e., 20ms or 40ms in this study) or different codeword length (i.e., 255 and 1023 in this study) does not incur noticeable difference in terms of tail latency, as shown in Fig. 8. The gain of applying local erasure coding improves as the soft sector read failure probability increases. In addition, the gain is relatively weakly dependent on the number of sectors being read. When we read 10k consecutive 4kB sectors, the use of local erasure coding can reduce the 99-percentile latency by 50.1% (τ_{retry}=20ms) and 67.2% (τ_{retry}=40ms) under the soft sector read failure probability p_h is 1×10^{-3}. The gain improves to 78.6% (τ_{retry}=20ms) and 88.1% (τ_{retry}=40ms) respectively as we increase p_h to 5×10^{-3}. When we read 100k consecutive 4kB sectors with p_h of 5×10^{-3}, the 99-percentile latency is reduced by 76.2% (τ_{retry}=20ms) and 86.6% (τ_{retry}=40ms). It should be pointed out that the per-sector read retry latency τ_{retry} strongly depends on how aggressively the HDD industry is willing to exploit the use of local erasure coding to push the magnetic recording technology scaling. Hence the results above mainly serve as the preliminarily estimation on the potential of applying local erasure coding to reduce HDD read tail latency.

4.3 Impact on Average Speed Performance

This work measures the impact on average system speed performance by running the following workloads in the benchmark suite HiBench 3.0: (1) Job based micro benchmarks Sort (*sort*) and WordCount (*wordcount*), which sorts and counts input text data generated by RandomTextWriter; (2) SQL benchmark *hivebench* that performs scan, join and aggregate operations, based upon the workload characteristics presented in [18]; (3) Web search benchmarks PageRank (*pagerank* and Nutchindexing (*nutch*); (4) Machine learning benchmarks Bayesian Classification (*bayes*) and K-means clustering (*kmeans*); (5) HDFS benchmark enhanced DFSIO (*dfsioe*); and (6) *terasort* a standard benchmark here sorts 1GB generated by teragen. All the experiments are carried out on one PC with a 3.30GHz CPU, 8GB DRAM, and 500GB 7200rpm HDD.

We integrated the software-based RS coding library into *ext4* in Linux kernel 3.10.102, and accordingly modified the I/O stack to incorporate the use of local erasure coding. Fig. 9 shows the measured throughput degradation of all the benchmarks when using software-based or hardware-based RS coding engine. Due to the absence of commercial CPUs with built-in FPGA, we estimated the results in Fig. 9(b) by adding delay into the I/O stack to mimic the effect of hardware-based RS coding, where we set the encoding/decoding throughput as 4GB/s. When running each benchmark, we randomly set one sector being read from HDD as a failure with the probability p_h, and accordingly carry out RS code decoding and issue additional HDD read if required by the decoding.

The average speed degradation is mainly due to the following three factors: (1) RS code encoding latency,

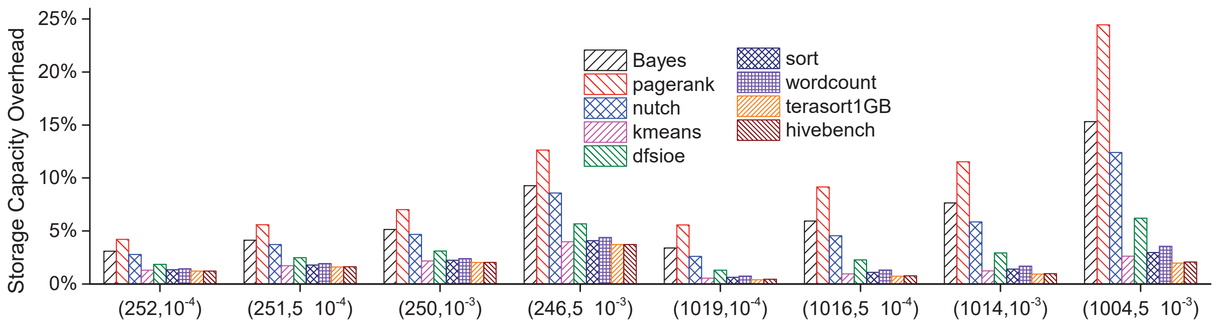


Figure 10: HDD storage capacity overhead under all the benchmarks when using the eight different RS codes. Each code is represented as (k, p_h).

(2) HDD latency for reading extra data required by decoding, and (3) RS code decoding latency. We note that, whenever RS code decoding is triggered for one group of $k+m$ sectors, we always first check whether the page cache contains some sectors in this group and only fetch un-cached sectors from HDD. Hence, the average speed performance degradation further depends on the data access patterns of each benchmark. Under the same value of p_h, a shorter codeword length (e.g., 255) incurs less degradation than a longer codeword length (e.g., 1023). This is because a longer codeword length results in a longer encoding latency and a higher probability of reading more HDD sectors during decoding. Compared with the software-based implementation, hardware-based RS coding implementation can reduce the average speed degradation by 60.6% on average. Even in the case of software-based RS coding implementation, the average speed performance degradation is not significant and can be (much) less than 10% when p_h is 1×10^{-3}.

4.4 Storage Capacity Overhead

Storage capacity overhead induced by the proposed design solution depends on both the coding parameters (i.e., k and m) and the file size distribution. We collected the file size distributions when running those HiBench 3.0 benchmarks, and accordingly estimated the storage capacity overhead as shown in Fig. 10, where each code is denoted using the parameters (k, m, p_h).

The results show that the storage capacity overhead increases as the sector read failure probability p_h becomes worse. This can be intuitively justified since, given the same target decoding failure probability, a worse p_h demands a stronger code with a larger coding redundancy. With the same code, different benchmarks have different storage capacity overhead as shown in Fig. 10, which is due to their different file size characteristics. The benchmarks (e.g., *pagerank*, *bayes*, and *nutch*), which have a large number of small files, incur relatively large storage capacity overhead. For example, in the case of *pagerank*, over 95% of its total storage usage is caused by small files with the size between 100kB and 1MB, and another 1.4% is caused by files smaller than 10kB. Because of the long codeword length (i.e., 255 and 1023 in this study), each small file is entirely protected by one shortened codeword with coding redundancy much higher than m/k. With p_h of 1×10^{-3} and codeword length of 1023, the storage capacity overhead is as high as over 12% for *pagerank*. In contrast, the other benchmarks (e.g., *kmeans* and *hivebench*), which are dominated by large-size files, have much less storage capacity overhead. For example, in the case of *hivebench*, over 99% of its total storage usage is caused by files larger then 100MB. As a result, with p_h of 1×10^{-3} and codeword length of 1023, the overall storage capacity overhead is only less than 2% for *hivebench*.

4.5 Impact of Fine-grained Data Update

Although data centers tend to avoid data update on HDD through the use of immutable data structure, it is still of practical interest to study the latency overhead incurred by data update. We note that only fine-grained data update (i.e., only a portion of data within one $(k+m)$-sector coding group is updated) is subject to latency penalty. As discussed in Section 3.4, we can use two different methods to carry out fine-grained data update. In order to update the data on HDD from $\mathbf{d} = [\mathbf{d}_1, \mathbf{d}_2]$ to $\mathbf{d}' = [\mathbf{d}'_1, \mathbf{d}_2]$, the first method reads $\mathbf{d}_2$ from HDD to directly re-compute the updated coding redundancy, while the second method reads $\mathbf{d}_1$ and old-version redundancy from HDD to in-directly compute the updated coding redundancy.

We carried the following experiments: With the codeword length of 255, we first encode and write 10GB data to a 7200rpm HDD. After clearing the OS page cache, we update l_u consecutive sectors at a random location within each codeword using either the first or second method.

We compare the total latency against the baseline without using local erasure coding. We repeat the experiments by setting l_u as 50 and 200, and Fig. 11 shows the measurement results when using different RS codes. The results show that the second method (i.e., in-direct coding redundancy re-computation) appears to be the better choice. In particular, for very fine-grained data update (i.e., update 50 sectors within 255 sectors), the second method significantly outperforms the first method.

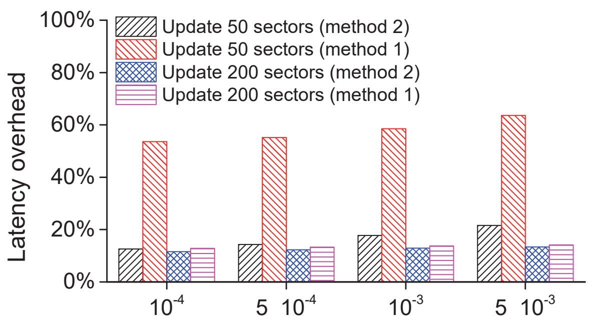


Figure 11: Measured fine-grained update latency overhead.

5 Related Work

Aiming to enhance storage systems by introducing coding redundancy across multiple sectors, this work shares the same nature as the widely used RAID and distributed erasure coding. Both RAID-5/6 and distributed erasure coding primarily target at tolerating catastrophic HDD failures, and typically employ erasure codes with relatively small codeword length (e.g., 12) and hence large coding redundancy (e.g., 20% to 50%). Recent work [3, 4, 19] presented specific erasure code construction techniques for RAID applications that can tolerate individual sector failures in addition to catastrophic HDD failures. However, RAID is being replaced by distributed erasure coding in data centers. Extensive research has been carried out on the construction of distributed erasure codes (e.g., see [9–11, 22]) and investigating/optimizing the system-level performance (e.g.,see [7, 12, 25, 29]). Nevertheless, the distributed nature makes it unsuitable for mitigating soft sector read failures of individual HDDs. Prabhakaran et al. [21] presented an Internal RObustNess (IRON) filesystem design framework that includes a variety of HDD failure detection and recovery techniques. Its transaction checksum technique inspired the implementation of journal checksum in *ext4* filesystem. It presented the metadata replication scheme, which is similar to the one being used in this work.

Researchers in the magnetic recording industry also recently investigated how intra-HDD erasure coding can complement with existing per-sector ECC (e.g., low-density parity-check (LDPC) codes) to improve the read channel performance. For example, the authors of [30] studied the effect of adding one parity check code across a number of 4kB sector inside one HDD. Nevertheless, intra-HDD realization of erasure coding is subject to several problems as discussed in Section 2.2.

6 Conclusions

This paper carries out an exploratory study on applying local erasure coding to facilitate technology scaling for data center HDDs. With finer track pitch, future HDDs are increasingly subject to areal density vs. read retry rate conflict. This is particularly serious for data centers that are very sensitive to bit cost and meanwhile cannot tolerate long HDD read tail latency. Aiming to alleviate such a dilemma, this paper investigates the potential and feasibility of using filesystem-level transparent local erasure coding to mitigate soft sector read failures. This paper presents the basic design framework and develops techniques to address two issues including unaligned HDD write and fine-grained data update. This paper further derives mathematical formulations for estimating the effectiveness on reducing tail latency, which has been quantitatively demonstrated through numerical analysis. To evaluate its impact on average system speed performance and demonstrate its practical implementation feasibility, we integrated this design solution into Linux kernel and carried out experiments using a variety of big data benchmarks. Its storage capacity overhead is also evaluated over various big data benchmarks. The analysis and experimental results demonstrate its promising potential and practical feasibility to address the bit cost vs. tail latency dilemma for future data center HDDs.

Acknowledgments

We would like to thank our shepherd Peter Desnoyers and the anonymous reviewers for their insight and suggestions that help us to improve the quality and presentation of this paper. This work was supported by the IDEMA/ASTC and the National Science Foundation under Grant No. CCF-1629218.

References

[1] *HiBench 3.0.* https://github.com/intel-hadoop/HiBench/releases.

[2] *Jerasure.* https://github.com/tsuraan/Jerasure.

[3] M. Blaum, J. L. Hafner, and S. Hetzler. Partial-MDS codes and their application to RAID type of architectures. *IEEE Transactions on Information Theory*, 59(7):4510–4519, July 2013.

[4] M. Blaum, J. S. Plank, M. Schwartz, and E. Yaakobi. Construction of partial MDS and sector-disk codes with two global parity symbols. *IEEE Transactions on Information Theory*, 62(5):2673–2681, May 2016.

[5] J. Blomer, M. Kalfane, M. Karpinski, R. Karp, M. Luby, and D. Zuckerman. An XOR-based erasure-resilient coding scheme. Technical report, Technical Report TR-95-048, International Computer Science Institute, August 1995.

[6] E. Brewer, L. Ying, L. Greenfield, R. Cypher, and T. T'so. Disks for data centers. Technical report, Google, 2016.

[7] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, et al. Windows Azure Storage: a highly available cloud storage service with strong consistency. In *Proc. of the ACM Symposium on Operating Systems Principles*, pages 143–157, 2011.

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

[9] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. J. Wainwright, and K. Ramchandran. Network coding for distributed storage systems. *IEEE Transactions on Information Theory*, 56(9):4539–4551, 2010.

[10] K. M. Greenan, X. Li, and J. J. Wylie. Flat xor-based erasure codes in storage systems: Constructions, efficient recovery, and tradeoffs. In *IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–14, 2010.

[11] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin. Erasure coding in windows azure storage. In *Proc. of USENIX Annual Technical Conference (ATC)*, pages 15–26, 2012.

[12] O. Khan, R. Burns, J. Plank, W. Pierce, and C. Huang. Rethinking erasure codes for cloud file systems: Minimizing I/O for recovery and degraded reads. In *Proc. of USENIX Conference on File and Storage Technologies (FAST)*, pages 251–264, 2012.

[13] A. R. Krishnan, R. Radhakrishnan, B. Vasic, A. Kavcic, W. Ryan, and F. Erden. 2-d magnetic recording: Read channel modeling and detection. *IEEE Transactions on Magnetics*, 45(10):3830–3836, Oct 2009.

[14] H. Lee. High-speed vlsi architecture for parallel reed-solomon decoder. *IEEE transactions on very large scale integration (VLSI) systems*, 11(2):288–294, 2003.

[15] F. Lim, B. Wilson, and R. Wood. Analysis of shingle-write readback using magnetic-force microscopy. *IEEE Transactions on Magnetics*, 46(6):1548–1551, Jun. 2010.

[16] S. Lin and D. J. Costello. *Error Control Coding: Fundamentals and Applications (2nd Ed.)*. Prentice Hall, 2004.

[17] K. Miura, E. Yamamoto, H. Aoi, and H. Muraoka. Estimation of maximum track density in shingles writing. *IEEE Transactions on Magnetics*, 45(10):3722–3725, Oct. 2009.

[18] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 165–178, 2009.

[19] J. S. Plank and M. Blaum. Sector-disk (sd) erasure codes for mixed failure modes in RAID systems. *ACM Trans. Storage*, 10(1):4:1–4:17, Jan. 2014.

[20] J. S. Plank and L. Xu. Optimizing Cauchy Reed-solomon Codes for fault-tolerant network storage applications. In *IEEE International Symposium on Network Computing Applications*, July 2006.

[21] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Iron file systems. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 206–220, 2005.

[22] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur. Xoring elephants: Novel erasure codes for big data. In *Proc. of the VLDB Endowment*, volume 6, pages 325–336, 2013.

[23] M. A. Seigler, W. A. Challener, E. Gage, N. Gokemeijer, G. Ju, B. Lu, K. Pelhos, C. Peng, R. E. Rottmayer, X. Yang, H. Zhou, and T. Rausch. Integrated head assisted magnetic recording head: design and recording demonstration. *IEEE Transactions on Magnetics*, 44(1), Jan. 2008.

[24] Y. Shiroishi, K. Fukuda, I. Tagawa, H. Iwasaki, S. Takenoiri, H. Tanaka, H. Mutoh, and N. Yoshikawa. Future options for hdd storage. *IEEE Transactions on Magnetics*, 45(10):3816–3822, Oct 2009.

[25] M. Su, L. Zhang, Y. Wu, K. Chen, and K. Li. Systematic data placement optimization in multi-cloud storage for complex requirements. *IEEE Transactions on Computers*, 65(6):1964–1977, 2016.

[26] D. Weller, G. Parker, O. Mosendz, E. Champion, B. Stipe, X. Wang, T. Klemmer, G. Ju, and A. Aian. A HAMR media technology roadmap to an areal density of 4 Tb/in^2. *IEEE Transactions on Magnetics*, 50(1), Jan. 2014.

[27] S. B. Wicker and V. K. Bhargava. *Reed-Solomon Codes and Their Applications*. IEEE Press, 1994.

[28] R. Wood, R. Galbraith, and J. Coker. 2-d magnetic recording: Progress and evolution. *IEEE Transactions on Magnetics*, 51(4):1–7, April 2015.

[29] L. Xiang, Y. Xu, J. Lui, and Q. Chang. Optimal recovery of single disk failure in rdp code storage systems. In *ACM SIGMETRICS Performance Evaluation Review*, volume 38, pages 119–130, 2010.

[30] S. Yang, Y. Han, X. Wu, R. Wood, and R. Galbraith. A soft decodable concatenated LDPC code. *IEEE Transactions on Magnetics*, 51(11):1–4, Nov 2015.