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

论文阅读报告

王傲

15300240004

操作系统

COMP130110.04

指导教师：谢志鹏

目录：

内容摘要：

本篇论文介绍了一种简单并且有效的应用于数据中心的硬盘（HDD）加速措施：使用文件系统级别的、对用户透明的本地纠删码（Local Erasure Coding）来降低硬盘重读（Read Retry）的次数，提高硬盘读取效率。这篇论文给出了基于 Reed-Solomon（RS）的本地硬盘纠删码的实施架构，并且推导出了用于评价实施纠删码前后系统效率的数学公式，提出了两个问题并给出解决方案，最后在 ext4 文件系统中进行了实验并获得结果。

本篇论文收录于 FAST 2017,作者为 Yin Li, Hao Wang, Xuebin Zhang, Ning Zheng, Shafa Dahandeh 和 Tong Zhang。

关键字：纠删码,Reed-Solomon,软性扇区读故障,尾延迟

1. 简介

随着 Flash Memory 的普及，民用市场上 SSD 的发展开始威胁到 HDD（Hard Disk Drive）的地位。然而，随着大量数据中心的兴建和基于云的计算设施的发展，对大容量、廉价的存储器材的需求迅速增加，这使得 HDD 成为数据中心非常好的存储设施。尽管如此，有研究指出，由于 HDD 自身的设计和结构，应用于数据中心的 HDD 仍然存在着一些问题需要解决。

这篇论文研究了应用于数据中心的 HDD（Data Center HDD）关于平衡大规模磁性存储设施部署和 HDD 工作特性的问题；具体的，考虑 HDD 的磁面密度（areal density）和重读频率（read retry rate）的矛盾：许多比较新的扩充 HDD 存储容量的技术本质上都是通过减小磁盘的磁轨间距来增大磁面密度，但是考虑到 HDD 磁头的机械运动，较小的磁轨间距会导致磁头对磁头偏移（head offset）非常敏感。如果偏移较大，就会引起 HDD 的重读以消除错误。所以，更小的磁轨间距会引起更大的重读率，导致更大的延迟；考虑到数据中心的大规模 HDD 部署，较高的重读率会导致无法容忍的尾延迟。[1]

所以，这就是这篇论文要解决的问题：如果存储在 HDD 中的数据本身就有一定的用于纠错（error correction）的冗余，那么当读取一个扇区发生失败时，HDD 就可以直接利用自身存储的冗余来计算（可能花费几十或几百微妙）而不是重读（几十或几百毫秒）。

[1] 少数远高于均值的延迟。

也许有人会问，独立磁盘冗余阵列（RAID）已经提供了这样的特性。然而，在数据中心里分布式的纠删码已经取代了 RAID，而利用分布式的纠删码来降低重读的影响会导致更高的开销（如网络延迟）。这篇论文着眼于在 HDD 和 JBOD（Just a Bunch of Disks）本地使用纠删码（local erasure coding），在与这个 HDD 或 JBOD 相连的服务器本地恢复无法读取或者读取错误的扇区。注意这种纠删码仅仅支持降低重读的花销，不支持灾难性的 HDD 崩溃的恢复。

在这篇论文中，HDD 无法解析通过正常操作获得的扇区的错误称为软性扇区读故障（soft sector read failure, 以下记为软性故障）。使用 p_h 标记实际情况中发生软性故障的概率，p_h 一般在 10^{-6} 或以下。与之相对，即便使用开销很大的重读也无法正确获取扇区的错误称为硬性扇区读故障（hard sector read failure），发生的概率非常低（10^{-14} 或以下）。当使用本地纠删码时，如果发生软性故障，HDD 并不马上开始重读，而是尝试使用本地纠删码来修复错误。只有当本地纠删码失效时，才使用重读。使用 p_s 标记本地纠删码失效的概率，我们的目标是在尽可能降低纠错码冗余程度的同时确保 $p_s << p_h$。为了降低编码冗余，这里必须使用较长的码字长度（codeword length,可能会横跨几百个 4KB 大小的扇区），所以这种编码适合于具有极大文件大小的系统（比如数据中心）。

这也带来了相关的问题：这种本地纠删码需要实施在应用程序层、操作系统层还是 HDD 内部？经过考虑，应该实施在操作系统层中的文件系统中，并且对于用户来说是完全透明的。这需要对文件系统进行修改。后续的实验在 ext4 文件系统上进行了这种尝试：具体的，对文件系统存储的数据和文件系统的元数据（meta data）进行了不同的操作，并同时提供了磁盘未对齐情况和细粒度数据更新情况的解决办法，同时给出了评估性能的数学公式。

实验结果显示，使用 Reed-Solomon 纠错码，当 p_h 为 10^{-3} 时，通过增加小于 2%的冗余可以消除 65%的 99 分位数的延迟（尾延迟）。

2. 背景和相关原理

2.1 磁性存储技术

业界和学界进行了大量扩充 HDD 容量的尝试，其中很大一部分通过显著降低磁轨间距来增大磁面密度。较小的间距会产生较大的轨间干扰（inter-track interference, ITI），会使 HDD 的读写磁头对磁头偏移非常敏感：

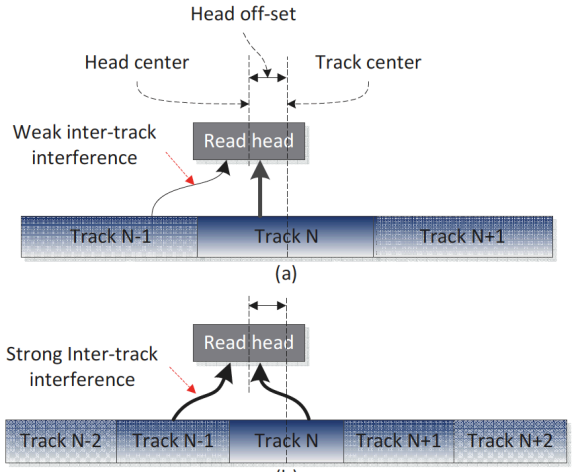

图 1 不同的磁轨间距产生的对磁头的影响

磁轨偏移是指目标磁轨中心和磁头中心之间的距离。磁轨中心和磁头中心完全对齐时产生读取失败的可能性最小，然而由于 HDD 自身的特性，盘面的旋转、外界震动等都有可能导致磁头偏离正常的移动，产生磁轨偏移；偏移的越大，磁头受到旁边磁轨的影响就越大，读取失败、需要重读的概率就越大。随着磁轨间距的减小，磁头对这种偏移会更加敏感，使得 HDD 的性能受到磁面密度和重读频率的矛盾的限制。

2.2 本地纠删码

本地纠删码旨在通过存储在 HDD 或者 JBOD 上的冗余码来降低重读发生的频率，从而降低时间开销。下面是不使用和使用本地纠删码的运行逻辑：

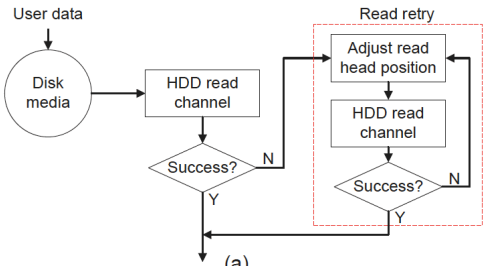

图 2 不使用纠删码，发生错误就调整磁头位置并重读

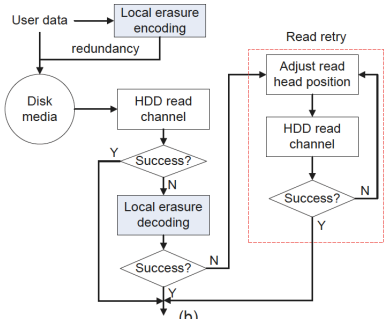

图 3 使用纠删码，发生错误先使用纠删码恢复，失败后再使用重读恢复

考虑到操作的便捷性、HDD 本身的特性以及某些数据对用户不可见，将本地纠删码实施在操作系统层的文件系统上。

2.3 纠错码选择

我们使用四个参数 {k, m, w, d} 来标记一个纠错码（error correction code, ECC）。每个码字用 m 个冗余段（redundant symbol）来保护 k 个用户数据段（data symbol），所以总的码字长度为 k+m。每个段有 w 个 bit，最小的码字距离为 d。对于每个线性 ECC，最小距离 d 服从 Singleton Bound: d <= m+1。其中，能够取到等号的 ECC 称为 MDS（Maximum Distance Separable）编码，能够达到最大的纠错强度。RS 码是一种最为知名的 MDS 编码。具有最短距离为 d 的 ECC 可以纠正最多$\lfloor\frac{d-1}{2}\rfloor$个 error 或者最多 d-1 个 erasure（这里 erasure 指能确定位置的 error）。一个为（k, m）的 RS 码可以保证在一个码字中最多能正确恢复 m 个 erasure。[2]

典型的 RS 码通过 Binary Galois Field（GF）实现。

3. 解决方案

3.1 基本架构

当实施具有透明性的本地纠删码时，要将文件系统的元数据（meta data）和用户数据区别对待。考虑到数据中心中极大的文件大小和粗粒度的文件访问模式，这里实施基于每个文件的长 RS 码（横跨几百个 4KB 大小的扇区，每个 RS 码字中的数据属于同一个文件）。基本架构如下：

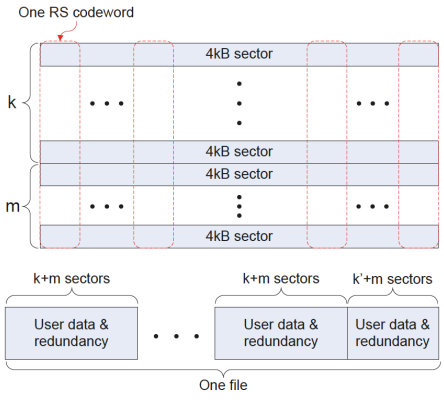

图 4 基于每个文件的本地纠删码架构

[2] 注：这里可以理解为 k 个数据项，增加了 m 个冗余的线性方程，方程的解与数据项相同。这样，只要丢失的数据项个数小于 m，就可以通过剩下的线性方程解出数据项（因为 k 个确定解需要 k 个方程解出）。

一个（k, m, w）RS 码字（红色区域）跨越 k+m 个连续扇区；每个扇区被分为数个段；每个 w-bit 的段（symbol）来自于一个扇区。为了简化表示，所有文件的（k, m, w）RS 码字参数都相同。用 N 标记一个文件中使用的 4KB 大小的扇区的数量，所以文件系统将一个文件划分为 $\lceil \frac{N}{k} \rceil$组，每组增加 m 个 RS 码冗余段。注意，最后一组可能存在 k'< k 的情况，使用（k', m, w）标记。最后一组的编码和解码方式与其他组相同，只不过将不足的地方填充为 0。

同样的，使用 p_h 标记发生软性故障的概率，使用 p_s 标记本地纠删码失效的概率（$p_s << p_h$）。当用户数据段的数量 k 确定时，我们要找到最小的 m，满足：

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

即类似一个二项分布。为阐明编码冗余率（m/k）与码字长度的关系，设定 p_s 为 10^{-8}，根据不同的 k 值计算 m 的最小值，结果如下：

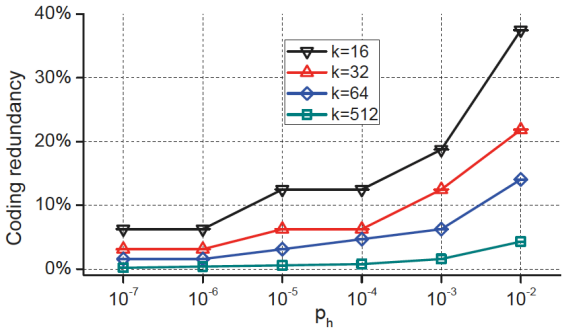

图 5 p_h 与 k 的关系；k 越大，冗余率越小

将文件大小视为一个随机变量，使用 g(x)标记文件大小的概率密度函数（PDF），x 为文件使用的扇区数量，则平均的存储冗余开销可以表示为：

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

这说明当文件大小增加时，空间上的冗余率会降低，并且每个文件的开销都是 m 个扇区，无论大小。

对于比较细粒度的文件系统的元数据，我们使用基于每个扇区的本地纠删码而不是基于每个文件。对于每一个元数据的扇区，我们分配 m'个连续的扇区来存储这个元数据扇区的 m'个备份。对于给定的 p_h 和 p_s，我们找到一个最小的 m'满足：

$$p_h^{m'} \leq p_s.$$

由于 p_h 不是很大，一个比较小的 m'就可以满足条件。所以这 m'个扇区很有可能在同一条磁轨上，不会产生显著的 HDD 访问延迟。由于 m'个扇区内容相同，我们标记第一个扇区为主扇区（lead sector）,其余 m'-1 个扇区为影子扇区（shadow sector）。所有文件系统结构中的指针指向主扇区，其余影子扇区仅仅在处理读错误时才使用。下图说明了在 ext4 文件系统中索引节点（inode）指向的内容：

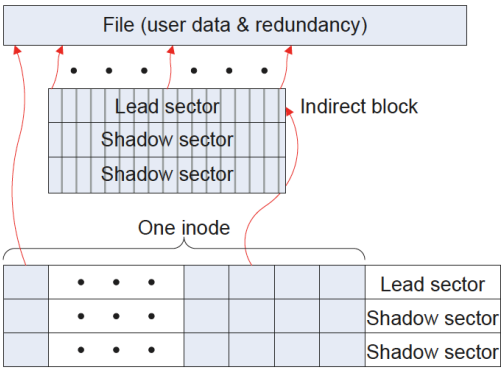

图 6 ext4 文件系统中基于复制的文件保护

为了在实际中实施上述对文件系统的设计，HDD 所在的服务器和 HDD 自身的固件需要被修改。除此之外，在实际中还可能会碰到以下两类问题：

1. 未对齐的 HDD 写操作（Unaligned HDD Write）：当写进 HDD 的用户数据没有完全填充一个或多个编码组的时候就会发生未对齐现象。理想状态下我们希望每一次对 HDD 的写操作都是与编码组的边界对齐的，比如我们可以在服务器内存中积累 k 条连续的用户数据段，经过纠删码编码后将 k+m 条数据写进 HDD。然而，实际中这样的条件并不能总是被满足，比如直接 I/O（direct I/O）和同步 I/O。即使是异步的 I/O，文件系统也可能会周期性刷新（periodic flushing）。这同样会导致数据与编码组的边界不对齐。

2.细粒度数据更新：尽管数据中心的应用绝大部分是粗粒度的文件访问（尤其是写数据时），细粒度的数据更新也可能会出现（比如更新十几个 4KB 大小的扇区）。在实际中，如果 RS 码非常长的话，对这种细粒度的更新的支持可能就不会十分高效。

这两个问题的解决方案会在后面给出。

3.2 尾延迟评估

考虑从 HDD 读取连续 N 个扇区的使用情景。用 T 标记读取 N 个扇区的延迟。考虑到纠删码和重读带来的额外延迟，把 T 视为一个离散的变量。因此，为了获得尾延迟（比如 99 分位数的延迟），使用 f(T)标记为 T 的概率质量函数（PMF），给定目标概率 P_{tail}，则应寻找尾延迟 T_{tail} 满足：

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

使用 τ_{retry} 标记重读的延迟（一次或几次磁盘旋转的时间），τ_u 标记 HDD 正常情况下读取一个扇区的延迟，忽略寻道延迟。考虑到实际情况（与硬盘生产厂家的联系），可以把软性故障视为独立同分布（IID）的随机变量，每次软性故障的发生是独立的。

所以，可以获得不使用本地纠删码时的延迟函数：

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

其中 t 是发生软性故障的次数。

在使用本地纠删码的情况下，定义 $l = \lfloor \frac{N}{k} \rfloor$，则 k'= N -l*k，k'是最后一组中不足 k 组的数据。由于解码可以与 HDD 的读操作同步进行，我们假设只有最后一次的解码造成了额外的开销。

下面分四种不同的情况讨论：

1. 前面 l 组都没有产生本地纠错码失效，后面 k'个用户数据扇区都没有产生软性故障：

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

2. 前面 l 组都没有产生本地纠删码失效，但是 e_0 个扇区在最后 k'+m 个扇区中产生读失败。用 $\tau_{dec}(e_0)$ 标记纠正 e_0 个扇区产生的延迟：

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

3. 前 l 组里，j 组产生了本地纠删码失效（e_1，e_2，e_3，... $e_j > m$），e_0 个扇区在最后 k'+m 个扇区中产生读失败。使用 p_i 标记在一个组里产生了 e_i 个纠删码失效的概率：

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

其中

$$p_i = \binom{k+m}{e_i} \cdot p_h^{e_i} \cdot (1-p_h)^{(k+m-e_i)}$$

4. 前 l 组里，j 组产生了本地纠删码失效，最后一组也产生了本地纠删码失效：

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

要估算尾延迟时，这四种情况每种都要考虑到。

3.3 改良未对齐 HDD 写操作

将未对齐 HDD 写规范化为以下形式：向量 $d=[d_1, d_2]$在一个码字中包含了 k 条用户数据，次级向量 d_1 和 d_2 包含了 k_1 和 k_2 条用户数据。未对齐写通常发生在这种情况：当文件系统必须将数据写进 HDD 时，只有 d_1 可以获得（即码字未被占满）。简单的处理方法就是直接将 k_1 和 m 条数据写进 HDD，尽管码字未被占满：

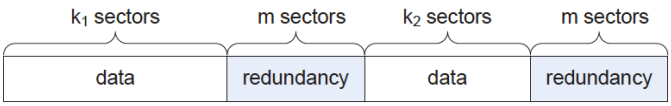

图 7 对未对齐写简单的处理方法；会造成过多冗余

这种方法会带来两个显著的问题：1. 文件系统需要在元数据中显式的记录 k_1 的长度，这会使文件系统的结构设计更加复杂；2. 存储的数据冗余会增加，降低系统的性能。

所以，对于未对齐写，仍然需要对固定长度的 k 使用同样的 RS 码。这里使用一种利用缓存的编码策略。考虑一般情况下 ECC 的生成：冗余码可以通过 $\mathbf{r}=\mathbf{G}\cdot\mathbf{d}$ 来表示，r 表示生成的 m 个冗余段，d 代表 k 个用户数据，G 表示 m*k 阶的用于生成冗余的生成矩阵。对于未对齐写的情况，定义两个长度为 k 的向量 $d^{(1)}=[d_1,O]$和 $d^{(2)}=[O,d_2]$，其中 O 代表 0 向量。因为 RS 码是通过 binary GF 生成的，所以有 $\mathbf{d}=\mathbf{d}^{(1)}\oplus\mathbf{d}^{(2)}$, 这里⊕表示异或操作。所以，此时编码过程可以写作：

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

定义 $r^{(1)}=G\cdot d^{(1)}$和 $r^{(2)}=G\cdot d^{(2)}$，我们有 $r=r^{(1)}\oplus r^{(2)}$。利用缓存的编码方式如下：

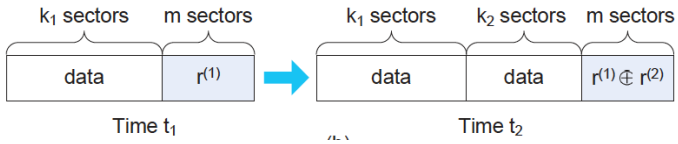

图 8 利用缓存的编码方式

文件系统首先写入 k_1 个数据扇区和存储着 $r^{(1)}$的 m 个冗余扇区到 HDD，同时操作系统在缓存中保存着 $r^{(1)}$。一旦剩下的 k_2 个数据扇区准备好了的话，文件系统就可以计算出 $r^{(2)}$，利用缓存中的 $r^{(1)}$计算出整体的冗余 $r=r^{(1)}\oplus r^{(2)}$。最后，文件系统将存储着 d_2 的 k_2 个数据扇区和存储着 r 的 m 个冗余扇区添加在 k_1 之后（$r^{(1)}$被覆盖）。由于 m 通常很小（5 或者 10）并且我们只需要存储中间产生的冗余（比如 $r^{(1)}$），这种策略并不会产生显著的缓存开销。显然，这可以推广到大于 2 的情况。

需要注意，当覆盖 $r^{(1)}$时产生了诸如电力故障这样的情况，d_1 就没有有效的本地纠删码保护了。不过这并不会造成数据完整性的问题，因为我们只是用本地纠删码来降低重读次数而没有修改原始数据。

3.4 处理细粒度数据更新

当发生细粒度数据更新时，操作系统将进行读-修改-写（read-modify-write）操作来更新 m 个冗余扇区。这会导致显著的系统性能下降，特别是细粒度更新是由同步写触发，同时在同一个编码组内的其他数据段并不在主机的内存中的情况。这里提出一种两段写（two-phase write procedure）的方法来解决这个问题。前面提到，本地纠删码只是为了解决重读带来的性能下降，所以某些数据暂时的不受纠删码保护也并不会带来数据完整性的缺失。换句话说，由于细粒度数据更新造成的 m 个冗余扇区不再可靠时，同一组内的 k 个数据扇区仅仅会遭受更高的重读风险而不会有数据丢失。所以两段写过程如下：当由于同步写造成细粒度数据更新时，文件系统首先服务于数据更新而不修改数据冗余；文件系统会在后台异步的修改 m 个扇区的数据冗余。如果细粒度数据更新不是由同步写触发，文件系统可以将这两步操作合并。

接下来的问题是如何更新数据冗余。为了简化问题，我们考虑以下情况：在一个有 k 条用户数据的码字中，将 $d=[d_1, d_2]$更新为 $d'=[d_1', d_2]$，并且没有更改的 d_2 部分不在主机内存中。为了将冗余数据 $r=G\cdot d$ 更新为 $r'=G\cdot d'$，我们有以下两种策略:

1. 直接用最简单的方式，即将 d_2 从 HDD 读入内存，然后通过重新计算获得 r'，最后将 r'写回 HDD。这需要从 HDD 读取 $k-k_1$ 个扇区。

2. 通过非直接的方式计算。定义 $d^{(1)}=[d_1,O]$，$d^{(1)'}=[d_1',O]$，$d^{(2)}=[O,d_2]$。定义 $r^{(1)}=G\cdot d^{(1)}$，$r^{(1)'}=G\cdot d^{(1)'}$，$r^{(2)}=G\cdot d^{(2)}$。原始冗余可以表示为 $r=r^{(1)}\oplus r^{(2)}$，同时可以表示为 $r\oplus r^{(1)}= r^{(2)}$。所以，r'可以表示为：

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

因此，可以通过原始的 d_1 和原始冗余 r 计算出 r'。这需要从 HDD 读取 k_1+m 个扇区。

同样的，这两种方法可以推广到大于 2 的情况。

4. 分析与实验

这里主要从两个方面评价上述解决方案的性能：1. 对尾延迟的提升：使用上面推导出的公式进行评估。2. 对平均系统速度的影响：这篇论文提出的 on-the-fly 的本地纠删码可能会带来系统在速度上的损失。这里使用大数据 benchmark 套件 HiBench 3.0 中的一系列 benchmark 来评估系统性能。

4.1 编码引擎的实现

设本地纠删码失效的概率 p_s 为 10^{-6}。p_h 为软性故障发生的概率，这里考虑四个不同的 p_h 的值：1×10^{-4}, 5×10^{-4}, 1×10^{-3}, 5×10^{-3}。设定目标码字长度为 255 和 1023，则 w 分别为 8 和 10。相应的 k 和 m 的值如下表：

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

这里使用了一个开源库 jerasure,将 RS 编码库融合进了 Linux kernel 3.10.102 下的 ext4 文件系统。编码和解码都使用了直接基于矩阵的运算而不是基于多项式的运算，这会最大化 CPU 中缓存的吞吐量。这里使用的实验平台为拥有 3.30GHZ 的 CPU 和 8G DRAM 的 PC，结果如下：

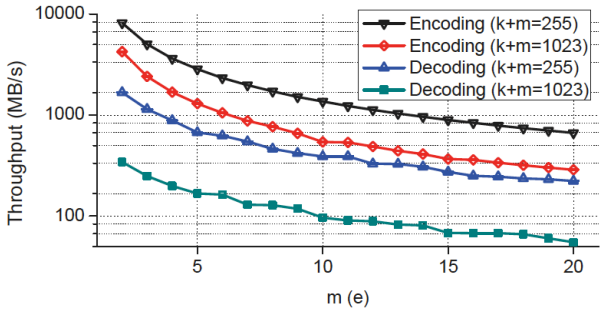

图 9 基于不同 m 的编码、解码吞吐量

使用 e 标记每个码字中被删去的段的数量，从结果可以看出，当码字长度一定时，m(e) 的增加会造成吞吐量的下降。这是因为计算中用到的矩阵的大小与 m(e)的大小成比例。实验结果同样说明，给定 m 和 e 时，解码的吞吐量小于编码的吞吐量。

除此之外，考虑到将 CPU 和 FPGA 融合到一起的发展趋势（比如一些 Xeon 处理器），这里同时尝试了基于硬件的 RS 编码引擎。由于存在充足的硬件级别的并行和高速的 CPU 与 FPGA 的交互，基于硬件的 RS 编码引擎能显著的增加吞吐量。这里使用 Berlekamp-Massey 算法构造了基于多项式的 RS 的编码器和解码器，以下是实验结果：

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

这里是 RTL（register transfer level）级的同步结果，所有实验均为具有 250MHz 的时钟频率的 4MB/s 的吞吐量。

4.2 尾延迟

使用 τ_{retry} 标记重读的延迟（一次或几次磁盘旋转的时间），τ_u 标记 HDD 正常情况下读取一个扇区的延迟，忽略寻道延迟。考虑使用 7200rpm 的 HDD，则 τ_u 为 33μs。根据实际情况，τ_{retry} 取 20ms 和 40ms 两个不同的值。实验结果如下：

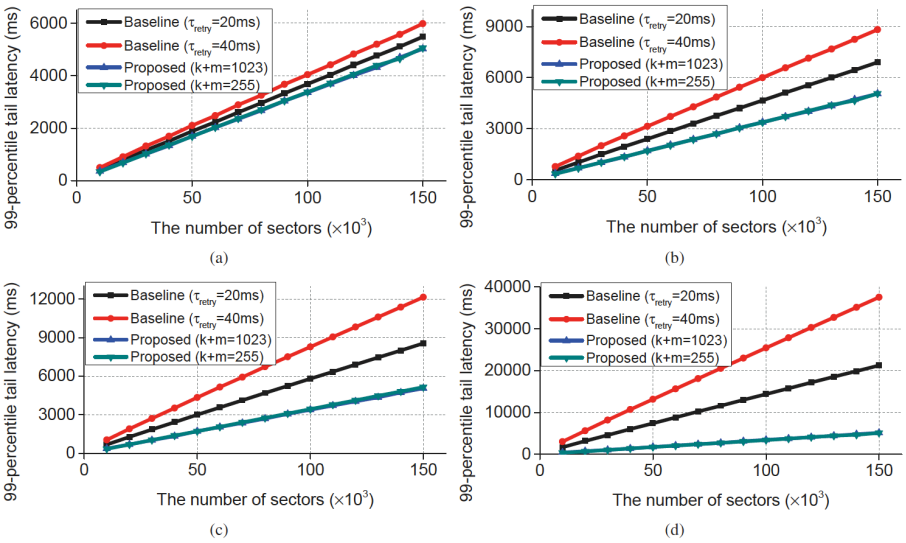

图 10 基于不同的 p_h 计算 99 分位数的延迟（尾延迟）

实验的结果说明了在高概率读失败场景下本地纠删码能显著降低 HDD 读的尾延迟。当读失败的概率上升时，使用本地纠删码带来的性能提升也会上升。除此之外，这种性能提升与读取的扇区数量关系不是很大。当读取 10k 个连续的扇区时，若取 $p_h=10^{-3}$，本地纠

删码能使 99 分位数的延迟降低 50.1%（τ_{retry}=20ms）和 67.2%（τ_{retry}=40ms）；若取 $p_h=5\times10^{-3}$，本地纠删码能使 99 分位数的延迟降低 78.6%（τ_{retry}=20ms）和 88.1%（τ_{retry}=40ms）。当读取 100k 个连续的扇区时，若取 $p_h=5\times10^{-3}$，本地纠删码能使 99 分位数的延迟降低 76.2%（τ_{retry}=20ms）和 86.6%（τ_{retry}=40ms）。

4.3 对平均速度性能的影响

这里使用 HiBench 3.0 中的一些工作负载来测试性能：1. 使用 micro benchwork Sort（*sort*）和 WordCount（*wordcount*），对 RandomTextWriter 产生的文本数据进行排序和计数；2. 使用 SQL benchmark *hivebench* 执行 scan, join, aggregate 操作；3. Web search benchmark PageRank（*pagerank*）和 Nutchindexing（*nutch*）；4. Machine Learning benchmark Bayesian Classification（*Bayes*）和 K-means clustering（*Kmeans*）；5. HDFS benchmark enhanced DFSIO（*dfsioe*）；6. terasort。所有实验在一台具有 3.30 GHz 的 CPU, 8GB DRAM 和 500GB 7200rpm HDD 的 PC 上进行。

下图是使用基于软件或硬件的 RS 编码引擎的实验结果：

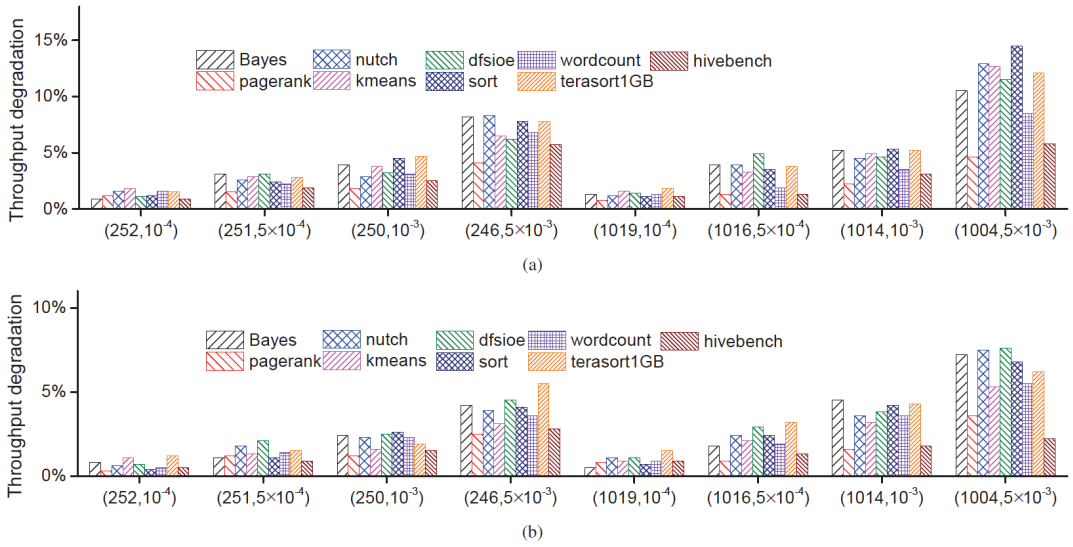

图 11 在不同 RS 编码下，不同 benchmark 造成的 HDD 存储开销；

(a)为基于软件，(b)为基于硬件

平均速度延迟是由以下三个因素造成的：1. RS 码编码延迟；2. 由于 HDD 需要读额外的数据来解码造成的延迟；3. RS 码解码延迟。这里需要注意，当需要解码时，我们只取回没有在缓存中的扇区，所以平均速度延迟还会受到每个 benchmark 访问数据的模式（data access pattern）的影响。当 p_h 相同时，一个较短的码字长度（255）会比一个

较长的码字长度（1023）产生更小的性能下降。这是因为较长的码字会导致更长的编码延迟和在解码时更高的读取更多扇区的概率。相比于基于软件的实施方法，基于硬件的 RS 编码能平均减少 60.6%的平均速度下降。即使是使用软件方法，当 p_h 为 10^{-3} 时，产生的速度性能下降不会超过 10%。

4.4 存储容量开销

由于增加本地纠删码造成的存储容量上的额外开销同时受到编码参数（比如 k 和 m）和文件大小分布的的影响。这里在运行 HiBench 3.0 的 benchmark 时收集了文件大小的分布，并计算了存储容量上的额外开销：

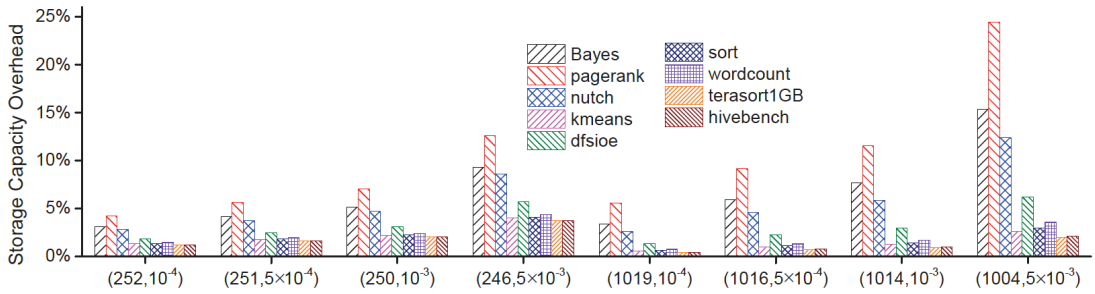

图 12 在不同 RS 码的情况下使用 benchmark 产生的存储容量开销

实验结果说明当 p_h 增大时，存储容量开销也会增大，这是因为一个更加大的 p_h 说明失败概率更大，需要更强（也更长）的冗余码来恢复。同时我们可以看出，有大量小文件的 benchmark（*pagerank*, *bayes* 和 *nutch*）存储容量开销也更大，而主要产生大文件的 benchmark（*kmeans* 和 *hivebench*）的存储容量开销会小很多。

4.5 细粒度数据更新的影响

需要注意，只有细粒度的数据 （比如一个 k+m 编码组的一部分）的更新才会产生延迟惩罚（latency penalty）。为了将 HDD 上的数据从 $d=[d_1, d_2]$更新为 $d'=[d_1', d_2]$，第一种方法是直接从 HDD 上读出 d_2 并计算新的冗余码，第二种方法是读出 d_1 和旧的冗余码，以一种非直接的方式计算新的冗余码。

在实验中，使用 255 的码字长，向一个 7200rpm 的 HDD 编码并写入 10GB 的数据。之后，清除操作系统的缓存页，使用第一种或第二种方式在一个随机选择的位置更新 l_u 个连续的扇区。将 l_u 设为 50 和 200，将时间延迟与不施加本地纠删码的时间进行比较，得到的结果如图：

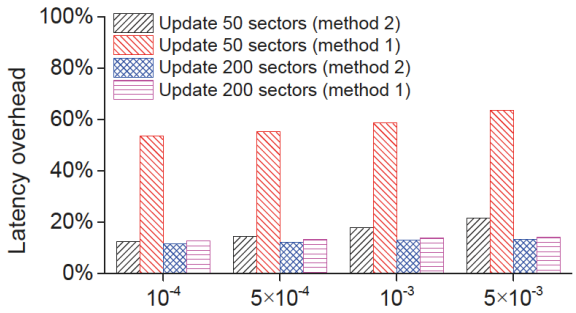

图 13 不同方式下的细粒度更新延迟

实验结果说明第二种方法（非直接计算）的效果会更好。在实际中，如果数据更新是非常细粒度的，第二种方法会比第一种在性能上好很多。

5. 结论

这篇论文研究了应用于数据中心的 HDD 关于平衡大规模磁性存储设施部署和 HDD 工作特性的问题。随着磁轨间距变的更加狭窄，未来的 HDD 会更加受限于 HDD 的磁面密度和重读频率的矛盾。这在对存储空间和 HDD 读延迟很敏感的数据中心中影响更加严重。这篇论文尝试使用文件系统级别的、对用户透明的本地纠删码来减少重读时延的影响，给出了实施本地纠删码的基本框架，提出了非对称 HDD 写和细粒度数据更新两个问题并给出了解决方案。这篇论文同时给出了评估降低尾延迟程度的数学公式。为了评估对平均系统速度的影响和实际应用的可行性，这篇论文使用了一系列 benchmark 在 Linux kernel 下的 ext4 文件系统上展开了实验,并对存储容量上的额外开销进行了计算和比较。实验的结果说明了本地纠删码的优良性能和在实际中应用于数据中心来降低尾延迟的可行性。

6. 相关工作

本篇论文的工作与 RAID 在思想上比较接近。RAID5、RAID6 和分布式纠删码都主要着眼于恢复被破坏的数据，所以使用了很小的码字长度（比如 12），因此会有比较大的编码冗余（20%-50%）。随着技术的发展，数据中心中分布式纠删码正在逐步取代 RAID，然而，分布式纠删码对于处理单独 HDD 的软性故障的效果不是很好。

7. 参考资料

1. Yin Li, Hao Wang, Xuebin Zhang, Ning Zheng, Shafa Dahandeh and Tong Zhang. Facilitating Magnetic Recording Technology Scaling for Data Center Hard Disk Drives through Filesystem-Level Transparent Local Erasure Coding. In *Proc. of USENIX Conference on File and Storage Technologies (FAST)*, pages 135 - 148, 2017.

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

3. Jialin Li, Naveen Kr.Sharma, Dan R. K. Ports, Steven D. Gribble. Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency. SOCC' 14 Proceeding of the ACM Symposium on Cloud Computing, pages 1-14