ZFS 文件系统中双容错编码性能的研究*

（Evaluating Encoding Performance of Double-Erasure Codes in ZFS）

张斌 眭聚磊 童健聪 王刚 刘晓光

Zhang Bin, Sui Julei, Tong Jiancong, Wang Gang, Liu Xiaoguang

（南开大学信息技术科学学院， 天津 300071）

（College of Information Technical Science of Nankai University, Tianjin 300071）

摘要：ZFS 是 Sun 推出的一款革新性的文件系统，它支持用户构建镜像、单容错或是双容错的软 Raid。其双容错编码方案采用的是 Reed-Solomon 编码（RS 码）。RS 码适用于任意系统规模及容错能力，因此广泛应用于多容错存储系统的构建。但由于是基于有限域运算，编码/解码时间复杂性差是其根本性的缺陷。ZFS 对写操作的处理采用的是聚合后追加的方式而非传统的覆盖方式，每次写操作都会进行一次编码计算。因此，编码计算性能是影响文件系统整体性能的重要因素之一。本文的工作是将 RDP 这一基于奇偶校验的双容错编码替代 Reed-Solomon 编码与 ZFS 相结合，以优化文件系统写操作的性能。我们设计了 cache 优化的 RDP 编码算法，在 ZFS 中进行了实现，并通过实验验证了这一方法的有效性。

Abstract: ZFS is an innovative file system invented by Sun. User can construct mirror, single fault-tolerant and double fault-tolerant Raid system on this file system. The double-erasure coding algorithm used on it is Reed-Solomon (RS code), which fits for systems in any scale and fault-tolerant., and is widely used in multi fault-tolerant system. But RS code based on calculating in Galois field has a high time complexity. The performance of coding is one of the most important factor of the whole performance of file system, because ZFS use appending writing strategy after writing request aggregated instead of traditional overwriting strategy and all the writing operation will be concerned with the coding calculation. An approach to improve the writing performance by combining between RDP double fault-tolerant encoding algorithm, instead of Reed-Solomon, and ZFS is shown. An algorithm to improve the RDP coding performance by using cache is designed and implemented in ZFS. At last, the experiments validate this opinion.

关键字：ZFS； 容错编码； RDP

Key words: ZFS Fault-tolerant Coding RDP

中图分类号：TP393 文献标识码：A

---

*作者简介：张斌（1987－），男，天津人，南开大学计算机系本科生，研究方向是网络存储。眭聚磊（1983-），男，南开大学计算机硕士，研究方向是分布式存储。童健聪（1988-），男，南开大学计算机硕士，研究方向是分布式存储与并行计算。王刚（1974－），男，南开大学计算机硕士生导师，计算机学会会员，会员号 E20-0006617M 刘晓光（1974－），男，南开大学计算机硕士生导师，计算机学会高级会员，会员号 E200008075M

通讯地址：300071 天津南开大学信息技术科学学院计算机系 Tel:022-23504780 E-mail:superpopb2b@126.com

1. 引言

随着信息化的不断发展，人们对存储的需求越来越高，这体现在容量、速度和可靠性等几个方面。几十年来，尽管 CPU 的计算速度得到了飞速的提升，但是磁盘的访问速度以及数据传输速度的提高远远比不上处理器的进步。同时，由于大容量的需求，单块磁盘往往是不够的。Raid[1]技术成为了人们关注的热点。一方面，通过聚合多块磁盘相连组成磁盘阵列，每个磁盘之间以并行的方式进行读写操作，既可以满足大容量存储的需求，又可以提高整体数据读写的速度。另一方面，磁盘阵列的数据以一定的编码方式组合，通过数据的冗余提高数据的可靠性，从而减少磁盘意外故障带来的损失。

与传统文件相比，ZFS 文件系统具有许多革新性的特点，能够显著提高文件系统的管理效率。Sun 骄傲的称其为“史上最后一个文件系统”[2]。ZFS 的特点主要包括：用存储池取代了卷管理器，文件系统的大小不再受限于设备，而是可以随着存储池的规模变化动态调整；引入事务性语义，避免了文件系统一致性检查带来的性能损失；128 位文件系统和元数据动态分配为 ZFS 提供了充分的存储空间和灵活性；基于镜像、RAIDZ（单容错 RAID5 的变形）和 RAIDZ2（双容错）提供的数据可靠性。其中，ZFS 的双容错冗余编码采用是与 Raid6 相同的 Reed-Solomon 编码[3]。对于文件系统来说，每一次的硬盘 I/O 都可以归结为对磁盘阵列条纹的读写。其中，每次写操作，都要涉及编码计算。而随着高性能存储介质如 SSD 的出现，以及应用对更高容错能力（更大的编码计算量）的冗余编码的需求，编码计算时间占写操作总时间的比重逐渐加大，因此编码的效率是影响文件系统的性能的重要因素之一。Reed-Solomon 编码是基于有限域的计算实现的，编码/解码计算复杂性较高。本文的主要工作是将新型容错编码 RDP 码[4]应用于 ZFS（基于 Fuse[5]的 zfs-fuse-0.5.0 版本）中，替代 Reed-Solomon 编码实现文件系统的双容错，从而提高文件系统的写性能。

2. 系统结构

2.1 总体结构

ZFS 文件系统整体结构分为用户部分和核心部分[6]。用户部分提供与用户的接口，包括用户输入的命令解析，将某个存储池虚拟为操作系统/dev 下的一个卷呈现给用户等等。

ZFS 的核心部分自上而下大致分为三层：接口层、事务对象层、存储池层。接口层为用户空间提供的数据接口和控制接口。事务对象层集合了数据管理单元、快照、属性处理、意图日志等的对象级别的实现。最底层的存储池层负责对存储设备进行管理，是事务对象层的各项策略与存储设备交互的通道。实现容错编码功能的 raidz 组件，就位于这一层。因此本文主要关注核心部分。

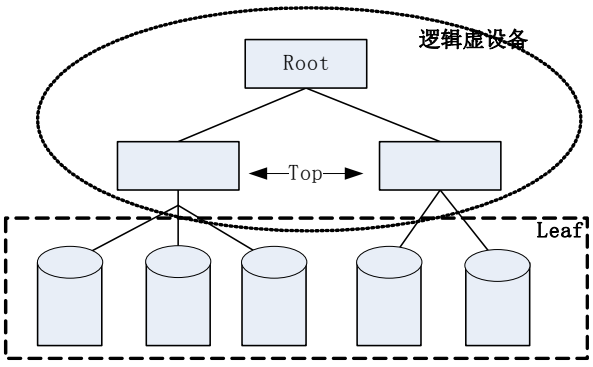

图 1 ZFS 存储设备组织方式

2.2 存储设备的组织方式

在讨论容错编码所处的层次之前，首先需讨论一下 ZFS 对于存储设备的组织方式。

ZFS 以存储池的方式组织存储设备，池中的文件系统不局限于单个设备，不同文件系统也可共享同一设备。池中所有设备都抽象成为虚设备，组成三个层次的树形结构。如图 1 所示，每个节点表示一个虚设备，每个虚设备的信息都存储在对应的 vdev 对象中。根节点称为 root 节点或者 root 虚设备；内部节点称为 top 节点或者 top 虚设备；叶节点称为 leaf 节点或者 leaf 虚设备。root 和 top 都是文件系统的逻辑虚设备；而 leaf 节点是物理虚设备，对应物理存储设备，可以是一个磁盘，或者是一个分区。容错编码应用于 top 层次，即一个 top 的多个孩子（leaf 设备）可组成一个磁盘阵列。

2.3 容错编码在文件系统中的位置

在 ZFS 中，任何用户 I/O 请求都被封装成为 zio 对象。请求被 ZFS 系统映射到虚设备进行进行实际的读写操作。如果请求访问的 top 节点的是 raidz 或者 raidz2 类型，则由 raidz 模块负责请求映射。图 2 显示了数据流向和层次关系，图中是一个由 7 个物理虚设备组成的双容错磁盘阵列。当写请求到达，raidz 将数据划分为数据单元(data unit)，并采用某种双容错编码方案计算(编码)出两个校验单元(parity unit)，然后将这些条纹单元（stripe unit，包括数据和校验）写入物理虚设备。对于读请求，只需映射到正确的物理虚设备，读取数据单元即可。当发生磁盘故障，校验单元被用来重构（解码）出丢失的数据单元。

ZFS维护两个数据结构来完成磁盘阵列的读写：

raidz_col 描述一个条纹单元的相关信息，包括条纹单元在条纹中的下标、I/O请求大小、I/O数据的指针、条纹单元的状态等。

raidz_map 描述一个条纹的信息，包括这个条纹中的条纹单元个数、状态、第一块数据盘在条纹中的下标，以及每个条纹单元的首址。

此外，raidz组件对外提供6个接口，其中和编码密切相关的接口如下：

I/O启动（vdev_raidz_io_start）接口：该接口生成一个raidz_map结构的实例。zio将记录的数据条纹化引用方式传递给此raidz_map对象。然后，对条纹中数据部分（而非校验盘数据）进行修改。如果是写操作，将调用处理阵列编码的函数，将校验单元写请求发送给子设备。如果是读操作，则从对应数据单元所在leaf虚设备进行读取。

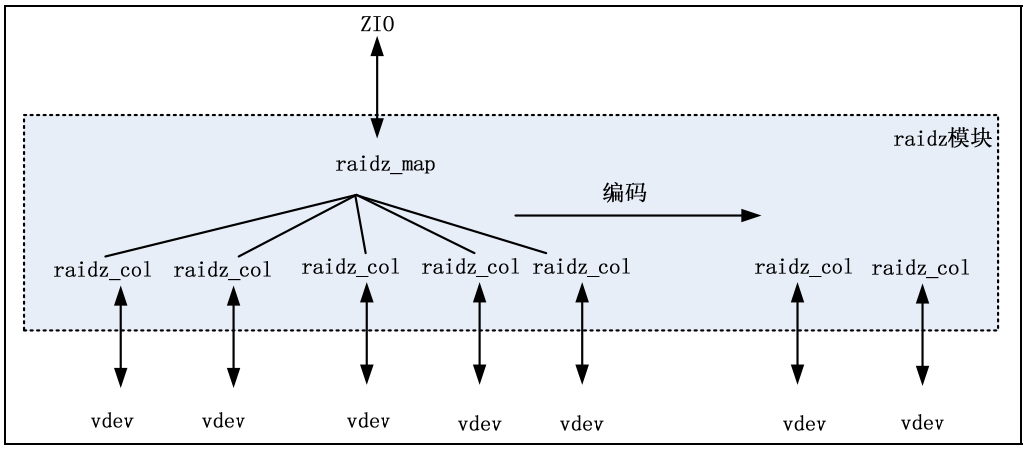

图 2 raidz 请求处理数据流向和层次关系

I/O完成（vdev_raidz_io_done）接口：该接口根据当前状态决定重构的策略：如果仅仅是校验盘损坏，调用编码函数重新编码即可；否则需要调用解码函数，对于双容错阵列，分为三种情况，即依

靠P盘解码（Q和一个数据盘损坏的情况）、依靠Q盘解码（P和一个数据盘损坏的情况）、依靠P/Q盘解码（两个数据盘损坏的情况）。如果任何方法均无法完成重构，将返回错误信息。如果收到的请求是写操作，或者发现需要重构，那么就调用底层函数，将条纹单元中的数据实际写入对应的leaf虚设备上。

3. 编码方法的改进

3.1 RDP码

RDP 码由 Peter Corbett 等人于 2004 年提出，是一种冗余最优的双容错编码。即，它只使用两个校验盘就实现了双容错能力。与 Reed-Solomon 码相比，RDP 码的校验计算只使用了奇偶校验操作，达到了最优的编码计算复杂性和接近最优的解码计算复杂性[4]。

标准的 RDP 码由 p+1 个磁盘组成（p 须为素数），其中 p-1 个为数据盘，另两个为校验盘。每个磁盘划分为 p 个块（packet），因此一个 RDP 编码系统可以用一个$(p\text{-}1)\times p$ 矩阵，这也是这类编码“奇偶校验阵列码”（parity array code）名称的由来。第一校验盘为水平校验盘，即，每个校验块由数据盘中相同偏移的数据块经异或运算而得。第二校验盘为对角线校验，即，同一对角线上的数据块和水平校验块经过异或运算得出对角线校验块。由于对角线校验依赖于水平校验，因此这类编码也被称为校验相关码。图 3 给出了 6 个磁盘的 RDP 码的示意图，每个块用一个数字表示所属的对角线校验组。注意，标记为“4”的几个块实际并未参与任何对角线校验组。公式（1）和公式（2）给出了 RDP 码的编码方案，其中 packet[i,j]表示第 j 个磁盘的第 i 个 packet。

需要注意的是，图 3 所示只是一个编码周期。在实际系统中，磁盘布局为此周期的重复。另一点值得注意的是，如果我们将 packet 实现为条纹单元，将导致较差的更新代价。即，一个条纹单元的更新一般会导致 3 个（而不是最优的 2 个）校验单元更新。但如果将编码周期的一列实现为一个条纹单元，则可达到最优的更新代价。从计算角度，也可达到最优的编码复杂性和接近最优的更新/解码复杂性。

Data Disk 0	Data Disk 1	Data Disk 2	Data Disk 3	Row Parity	Diag. Parity
0	1	2	3	4	0
1	2	3	4	0	1
2	3	4	0	1	2
3	4	0	1	2	3

图 3 条纹示意图（p=5）

$$packet[i,p-1]=\bigoplus_{j=0}^{p-2} packet[i,j],(i=0,1,...p-2) \quad (1)$$

$$packet[i,p]=\bigoplus_{\substack{j=0\\ j\neq i+1}}^{p-1} packet[k,j],(k=(i-j+p)\bmod p, i=0,1,...p-2) \quad (2)$$

3.2 编码算法

对于单容错的奇偶校验码（RAID4/5），编码计算直接将数据字进行异或运算，即可得到校验字。由公式（1）（2），直观的编码算法也可以校验字为中心，即按顺序计算水平校验字 packet[0,0] ~ packet[p-2,0]，然后以类似顺序计算对角线校验数据。但与单容错码不同的是，在单容错系统中，每个数据字都只参与一个校验组，在编码计算过程中只被访问一次。而在 RDP 系统中，每个数据字参与两个校验组，如果以校验字为中心进行编码计算，每个数据字会被访问两次。我们知道，在计算机系统

中，访问内存数据时，首先会将数据读入 cache，然后才会由 CPU 进行处理。因此，在以校验字为中心进行 RDP 编码计算时，若条纹单元较大，每个数据字会两次进入/换出 cache，导致对内存带宽的较大压力。

文献[7]提出了以数据字为中心的策略，即，每读入一个数据字，将它异或到所属的两个校验字。这样，每访问一个数据字，它所涉及的运算一次性完成，每个数据字只进入 cache 一次，极大地提高了 cache 的使用效率。由于基于奇偶校验的单/双容错码的计算复杂性较低，可以认为其编码运算是一种数据密集型运算，因此编码性能主要取决于访存性能。以数据字为中心的计算方式由于可以更为有效地利用 cache，带宽需求更低，因此可以明显提高编码性能。对 RDP 码，需要注意的是，当所有数据字处理完毕后，还需访问对角线校验字所依赖的水平校验字一次，完成对角线校验的最终计算。

在以数据字为中心进行编码计算的基础上，对于数据字参与运算的顺序，同样有两种选择。一种是行优先的方式：首先处理第一个数据盘的第一个 packet，接着处理第二个数据盘的第一个 packet，…，完成水平校验盘的第一个 packet 之后，再处理第一个数据盘的第二个 packet，依此类推，直至水平校验盘的最后一个 packet 处理完毕。另一种是列优先的方式，即首先顺序处理第一个数据盘的所有 packet，接着处理第二个数据盘的所有 packet，依此类推，直至水平校验盘。由于每一列（磁盘）的数据在内存中是连续的，相比之下，列优先的方式可以更好地利用 cache 的预取机制，掩盖内存至 cache 的延迟，因此当条纹单元较小时具有更好的性能。

3.3 RDP码在ZFS中的实现

传统磁盘阵列对小数据写的处理方式为“读－修改－写”策略（Read-Modify-Write，RMW），即，首先读取要更新的数据单元的旧有内容和校验单元的旧有内容，然后结合更新数据计算出校验单元的新内容，最后将更新数据和校验单元的新内容写入磁盘。对于小数据，RMW 策略明显优于重构写策略（Reconstruct Write，读取不更新的数据单元来和更新数据一起计算出校验单元的新内容），但仍然需要 4 次磁盘操作才能完成一次写请求。而且，当写操作过程中发生系统崩溃，就有可能造成同一条纹中数据单元和校验单元的不一致。

针对传统磁盘阵列的这些缺点，ZFS 采用了写请求聚合策略。即，推迟写请求的处理，将多个写请求组合在一起，形成整条纹写入磁盘，这样即可解决小写性能缺陷。但这一策略需要解决两个问题。首先，传统的用户地址到磁盘阵列地址的映射为固定映射方式，数据更新应覆盖固定地址的旧数据。这样，多个写请求的覆盖地址可能是不连续的，则无法组合为整条纹。为此，ZFS 采取了追加写方式。即，新数据并不覆盖旧数据，而是写入新分配的磁盘空间中，而旧数据占用的空间会适时回收。这样，只要将多个写请求分配到同一条纹的地址中，即可实现整条纹写。此外，追加写还可有效解决写操作中系统崩溃造成的不一致问题。另一个问题是，若系统负载较轻，可能较长时间也无法组合出足够大小的写数据量。为此，ZFS 采用了可变条纹设计。即，条纹长度（条纹单元数）与条纹单元大小均是可变的，根据写请求聚合情况动态确定。

为了满足上述处理策略的需求，RDP 需要在标准编码结构的基础上加以变化。首先，RDP 要求条纹单元大小为 p-1 的整数倍，其中 p 为素数。ZFS 在请求聚合之后，形成的条纹单元大小为磁盘扇区大小的整数倍，在 512B～128KB。因此，ZFS 很可能无法满足 RDP 对条纹单元大小的需求。假定 ZFS 组合构成的条纹单元大小为 S，我们先利用公式（3）将其调整为 S'。

$$S' = \left\lceil \frac{S}{p-1} \right\rceil \times (p-1) \qquad (3)$$

即，将其调整为不小于 S 的最小的 p-1 的整数倍。这样，即可满足条纹单元为 p-1 的整数倍的需求。但是，假定条纹中数据单元的数目为 k，则条纹总大小为 kS'，而用户写请求总数据量只有 kS。我们尽量均匀分布数据：

$$s_i=\begin{cases} S' & 0\le i<k-1-\lfloor k(S'-S)/P\rfloor \\ S'-P & k-1-\lfloor k(S'-S)/P\rfloor\le i<k-1 \\ S'-k(S'-S)\bmod P & i=k-1 \end{cases} \quad (4)$$

其中 P 为 packet 大小，即 $S'/(p\text{-}1)$。于是，一个实际的布局即可能如图 4 所示，部分磁盘存储大小为 S'的完整条纹单元，其余磁盘存储的数据量不足 S'，不足部分在图中以阴影表示。在编码/解码运算中，阴影区域可视为全 0，因为 0 对异或运算的结果不会有任何影响。实际上，对于阴影区域完全不必进行实际的磁盘读写，直接在内存中将其设置为 0 即可。可以看到，这个策略尽量均匀分布数据，未将全部“不足”部分分布在最后一个数据盘，这样可以使不同数据盘的读写负载更为均衡。对于校验盘而言，显然应该是完整条纹单元大小。应该注意的是，实际上 ZFS 组合构成的条纹中，条纹单元就可能不是一样大的，实际调整方式更为复杂一些，但基本思想是一致的。

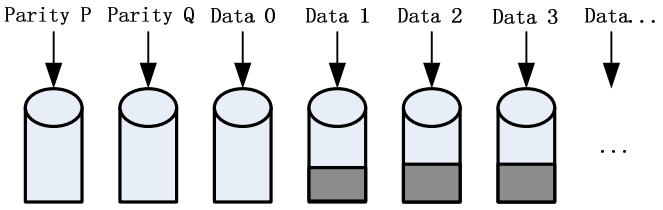

图 4 ZFS 条纹数据分布情况示意图

另一个需要解决的问题是，标准 RDP 编码结构要求条纹长度为 $p+1$，p 为素数。而系统实际配置中，构成一个磁盘阵列 top 节点的 leaf 节点数可能不是素数规模。而且，ZFS 采用写请求聚合整条纹写的策略，实际条纹长度还可能小于 leaf 节点数。我们可以采取“编码缩减”技术，来使 RDP 码适应变化的、非素数的条纹长度。即，预先选定一个较大的 p，使 $p+1$ 大于可能出现的实际条纹长度。对于 ZFS 确定的实际条纹长度 g（$<=p+1$），删除 g 个数据单元（假定为全 0），显然，这样不会影响编码的容错能力和编码/解码算法。

在 ZFS 中，条纹中排列在后的单元为“不足”单元。因此，我们将两个校验单元放置在条纹前端，保证校验单元规模为 S'，如图 4 所示。RDP 编码方案修改为：

$$packet[i,0]=\bigoplus_{j=2}^{n} packet[i,j],(packet[i,j]\in E,i=0,1,...p-2) \quad (4)$$

$$packet[i,1]=\bigoplus_{\substack{j=2\\ j\neq i+3}}^{n} packet[k,j]\underset{m\neq p-1}{\oplus} packet[m,0],(packet[k,j]\in E,i=0,1,...p-2) \quad (5)$$

（其中 $k=(i-j+2+p)\bmod p, m=i+1$）

其中 E 表示有效数据区，即图 4 中浅色区域。编码算法采用数据字为中心，列优先的策略。

4 实验验证

4.1 实验环境

为了验证 RDP 编码的有效性，本文在 ZFS 上实现了一个基于 RDP 的双容错编码组件，并通过实验来测试其性能。具体的实验环境如表 1 所示。测试工具采用的是 iozone，版本是 3.279-1.el3.rf.x86_64，测试文件大小 3GB。在实验中，p 取 17，这样还可保证 S 可被(p-1)整除，所有条纹单元均为同样大小。

表 1 软硬件实验环境

	配置
CPU	Intel(R) Xeon(TM) 3.00GHz×2
内存	1GB
硬盘	SAS 73.2GB×6
操作系统	Linux Red Hat AS 5
文件系统	Zfs-fuse-0.5.0
Fuse	版本 2.7.3

4.2 测试结果

Reed-Solomon 和 RDP 编码的不同之处仅仅在于校验计算，从磁盘读写角度看则无任何差异。因此，我们仅测试采用两种编码情况下 ZFS 的写性能，以此来考察编码计算性能对系统整体性能的影响。

Reed-Solomon 和 RDP 编码都是使用两个校验单元来提供双容错数据冗余能力。从理论上分析，采用上述两种双容错编码的由 N 个硬盘构成的存储系统，它的最大 I/O 性能应该不高于由 N-1 块硬盘组成的 Raid5 系统度。考虑到双容错系统的第二块校验盘的编码计算所需时间，双容错系统的实际 I/O 性能应该低于由 N-1 块硬盘组成的基于 RAID5 单容错系统。

图 5 所示的实验结果验证了这一点。我们使用 6 块相同型号硬盘，分别基于 Reed-Solomon、RDP 和 RAID5 组成存储系统，并利用 iozone 来测试系统的写操作性能。由于使用的 ZFS 大部分代码在用户态，需频繁进出内核，所以总体性能并不是很好。而且用户态的程序容易受系统的影响，因而测试结果的波动较大。尽管如此，从图 5 的结果中还是容易看出，使用 RDP 编码能够显著提升双容错文件系统的写操作性能。与 Reed-Solomon 编码相比，性能最多提高 6%以上。

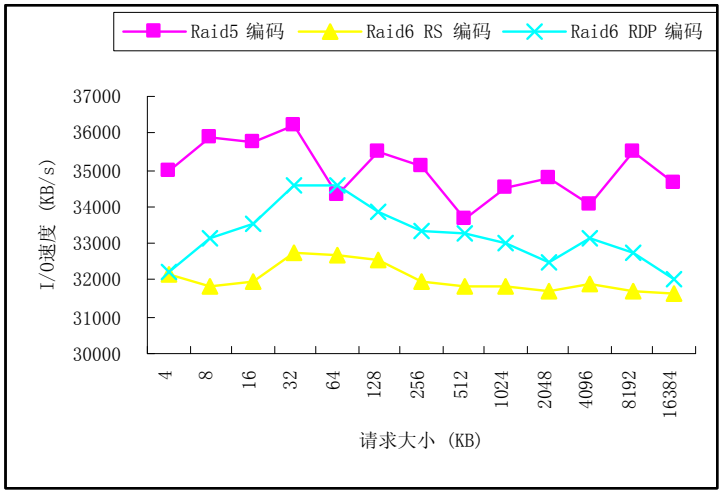

图 5 性能比较

5 结束语

本文提出了利用 RDP 编码替代 Reed-Solomon 编码实现 ZFS 文件系统的双容错，来提高系统写性

能的方法。实验结果表明，这一方法确实能够明显提升双容错 ZFS 文件系统的性能。在今后的工作中，我们将对 ZFS 在其他方面如事务管理和快照技术等的工作进行研究和分析[8]，希望能够提出进一步的优化方法。此外，对应用更为广泛的 Linux 内核磁盘阵列模块采用类似的技术进行优化，也是未来重要的研究内容之一。

参考文献

[1] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, David A. Patterson. RAID: high-performance, reliable secondary storage[A] In ACM Computing Surveys[C],1994, pp. 145-185.

[2] Bill Moore, ZFS - The Last Word in File Systems[EB/OL] http://www.sun.com/software/solaris.

[3] J. S. Plank. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems[A]. In Software: Practice and Experience[C], 1997, pp. 995-1012.

[4] Peter Corbett, Bob English. Row-Diagonal Parity for Double Disk Failure Correction[A]. In: Proceedings of the 3rd USENIX Conference on File and Storage Technologies [C].2004, pp. 1-14.

[5] Lonczewski, F., Schreiber, S. The FUSE-System:an Integrated User Interface Design Environment[A]. In Proceedings of Computer Aided Design of User Interfaces[C], Namur, Belgium: 1996, pp. 37-56.

[6] admin, ZFS Source Tour[EB/OL]2009，http://hub.opensolaris.org/bin/view/Community+Group+zfs/source.

[7] Jianqiang Luo, Lihao Xu, James S. Plank, “An Efficient XOR-Scheduling Algorithm for Erasure Codes Encoding,”[A], In: Dependable Systems & Networks, 2009. DSN '09. IEEE/IFIP International Conference on[C]，2009, pp. 504-513

[8] Rodeh, O., Teperman, A. zFS - a scalable distributed file system using object disks[A], In Mass Storage Systems and Technologies, 2003. (MSST 2003). Proceedings. 20th IEEE/11th NASA Goddard Conference on[C], 2003, pp. 207-218.