海量存储　课程作业

元数据改进及 ext4 文件系统针对尾延迟的优化

学院：计算机学院

专业：计算机技术

姓名：闫悦菁

学号：2017282110303

【摘要】

本文是在对于两篇 fast 会议论文的理解基础上，进行了相关的总结及思考。第一篇为 The Composite-file File System:Decoupling the One-to-One Mapping of Files and Metadata for Better Performance，说明了传统的文件系统中逻辑文件与物理元数据通常是一对一的映射，这种方法在现有文件系统下并没有很高的效率。于是有一种新的方法即为打破一对一映射，利用一定规则将关联性强的文件的元数据打包为一个节点，节省了空间，提升了效率。第二篇为 Reducing File System Tail Latencies with Chopper。利用 chhopper 工具，具体探讨了在 ext4 文件系统中导致长的尾延迟的因素，识别其中的四个主要问题后，精准定位其代码，通过修改策略来讲尾部的大小减少一个数量级，产生满意的文件布局，减少延迟。

关键字：元数据；一对一映射；ext4；尾延迟

第 1 章　概述

随着科技的发展及时间的推进，现在的社会变成一个高速发展的社会，科技发达，信息流通，人们之间的交流越来越密切，生活也越来越方便，在这个高科技时代下，大到企业小到个人，需要存储的数据越来越多。当数据变多，不能只是无休止的增多存储器，而是从对于数据的存储和利用中来增强功能性。

元数据是对于文件数据进行描述的数据，一般来说元数据与逻辑文件是一对一的关系。一般来讲，文件系统中，对小文件的访问比较频繁，而每个文件都有自己的元数据，这就需要很大空间来存储元数据。而很多文件的元数据是相似，甚至有相同的部分，冗余的数据也会对存储造成负担。于是在 2016fast，即 14th USENIX Conference on File and Storage Technologies 会议上，Florida State University 的几位学者提出了复合文件的文件管理系统，打破了原有逻辑文件与元数据一对一的映射方式，在 Linux ext4 文件系统下进行了实验验证。

开发和运维并发系统的过程中，可能会有这样的情况，明明系统已经调优完毕，该异步的异步，该减少互斥的地方引入无锁，该减少 IO 的地方更换引擎或者硬件，该调节内核的调节相应参数，然而，如果在系统中引入实时监控，总会有少量响应的延迟高于均值，我们把这些响应称为尾延迟（Tail Latency）。对于大规模分布式系统来说，尾延迟的影响尤其严重。尾延迟可能是程序设计本身导致的毛病，但是，即便程序设计完全无误，尾延迟依然可能存在。硬件，操作系统本身，都可能导致尾延迟响应，例如：主机系统其他进程的影响，应用程序里线程调度，CPU 功耗设计等等。研究表明，本地文件系统是产生 tail latency 的关键因素，发现在目前大多数分布式文件系统的核心，本地文件系统如 Linux ext4，XFS 和 btrfs，是现在可扩展存储的构建块。因此，一旦本地文件系统的性能出现问题，那么构建在它上面的分布式文件系统的性能将受损。来自 Department of Statistics University of Wisconsin–Madison 的学者们在 2015fast 上发表自己的成果，提出了一个名为“Chopper”的工具，它主要关注于块分配，因为这是造成文件系统异常的关键因素。他们在 Linux ext4 文件系统上完成相关实验，发现了造成 high-latency 的原因，用 Chopper 来减少文件系统的尾延迟。通过这个实验精准定位并删除了 ext4 的四个布局问题，修复了文件系统。

以下将分别讲解复合文件系统的构成及操作方法，以及对文件系统进行控制变量法查找影响因素的实验。并对两者的相似性进行比对以找到今后问题的研究思路，及工作展望。

第 2 章　元数据及复合文件系统

2.1　背景简介

我们先来了解一下 Metadata，中文翻译为元数据。它的标准定义是描述其它数据的数据（data about other data），或者说是用于提供某种资源的有关信息的结构数据（structured data），其使用目的在于：识别资源；评价资源；追踪资源在使用过程中的变化；实现简单高效地管理大量网络化数据；实现信息资源的有效发现、查找、一体化组织和对使用资源的有效管理。简言之，元数据描述了文件的各种属性。比如用户是否有权限打开或者对文件进行读写操作，文件最新的一次更新是什么时候等等。

关于文件系统，传统的文件系统中，逻辑层面上的文件和物理层面上的元数据之间，都是一一对应的关系。即，每个文件只与他自己的索引节点相关联。这种映射方式是可取的，因为数据元结构是根深蒂固的数据结构，许多存储部件和机制都依赖这种结构，比如 VFS API，预读取和数据元缓存。

但是，这种一对一映射方法导致它失去了某一类优化的可能性。我们可以观察到，用户一般不会访问单个文件，而是一系列文件一起访问，这一系列文件通常较小（小于 32 字节）或者有相似的元数据 （相似元数据是指有些文件的属性相同）。现在的文件系统有很多的问题，第一是小文件的频繁访问问题。对桌面文件系统的内部研究证明，>80%的文件访问涉及到的文件大小不超过 32 字节，在磁盘上访问小文件时，40%的时间是用来访问元数据。 第二，现实应用中，一个传统文件是与其物理数据元联系在一起的，数据元中记录了诸如文件块的位置、访问权限等信息。然而，许多文件有相似的文件属性。因为文件所有者的数量、权限模式等可能取值是有限的。所以存在大量的元数据冗余信息，比如典型工作站的元数据压缩比例高达 75%。这么高的冗余就会让我们去思考有没有合并的可能。第三，多个文件倾向于被同时访问，但是我们采用一一映射方式，效率较低。 第四，预先读取数据会产生限制，对每一个文件进行访问时，都会带来很大的开销，比如访问 10 个 10KB 的子文件相比于访问一个 100KB 文件来说，延迟增加了 50%。

2.2 复合文件系统

复合文件的文件系统（Composite-file File System，CFFS）就是为了克服以上问题。

传统的文件是将文件单独放置，文件与文件之间的关联性不大。如果将一次访问的多个小文件合并在一起，就可以提高系统的性能。这也就是本文设计实现的复合文件系统——CFFS，这个系统解除了逻辑层面上的文件和物理层面上的元数据之间的一一对应关系，它引入一个内部物理表示，称为复合文件，这个文件会保存经常一起访问的小文件的内容，它们（多个小文件）共享一个索引节点，从而实现从文件到元数据的多对一的对应关系。

传统的文件系统元数据节点如图 1 所示，而复合文件的文件系统信息节点如图 2 所示。

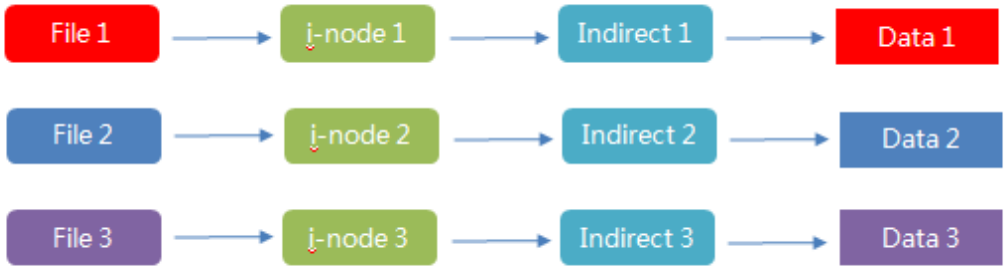

图 1.传统文件系统

图 2.复合文件的文件系统

由上图我们可以看到，传统的文件系统，每个文件对应一个索引节点，文件 1 对应索引节点 node-1 和 data 1，文件 2 对应索引节点 node-2 和 data 2，每个文件具有各自对应的属性值。而复合文件系统，首先分配一个索引节点 node-C，然后将各个子文件的数据部分连接起来作为复合文件的数据。 各个文件的属性信息统一加到该

节点的属性信息中。 复合文件会在其扩展属性中记录 1.复合文件偏移量 2.每个子文件的大小 3.被删除的重复 inode 信息 等信息。

在复合文件中有如下一些性质及操作：

复合文件的权限是所有文件权限的和。因此在检测访问权限时，首先检测复合文件的访问权限，然后再检测目标子文件的访问权限。比如 A 只读，B 可写，那么复合文件权限就是读写，但具体在打开 A 文件时，扩展属性中的只读权限就不允许写入操作。

I-node 内容重构：重复删除的子文件 i-node 实时重建。默认情况下，除非在扩展属性中另行指定，否则子文件的 i-node 字段会继承复合文件的 i-node 字段的值。

时间戳：每个文件操作都会更新单个子文件和复合文件的时间戳。 但是，在检查期间（例如统计系统调用），我们返回子文件的时间戳。

i-node 命名空间：对于大于阈值 X 的 i-node 数字，较高的 N 位扩充 0 被用作复合 i-node 号码，较低的 M 位保留用于子文件 ID。我们将这个范围的 inode 号码称为 CFFS 唯一 ID（CUID）。如图 3 所示。

Upper bits	Lower bits
00	**0**
00	**1**
01	**0**
01	**1**
10	**0**
10	1
11	**0**
11	1

图 3、iNode 命名

对子文件进行的操作主要有，打开、关闭、添加、删除、读取、写入、空间压缩等。

当进行打开某个子文件操作时，先找到复合文件的入口点，然后再根据目标子文件的偏移量找到数据位置。 关闭子文件时，则需

要关闭整个复合文件。 添加子文件时，将文件数据附加在原来数据后面。 删除子文件时，则将相应的数据释放。

读取文件时，根据目标子文件的位置偏移找到数据起始位置，然后根据子文件的大小读取相应的长度。 写入数据时，首先根据目标子文件的位置偏移找到写入数据的起始位置，然后将相应的数据写入。如果没有足够空间，就把子文件移至复合文件的末尾，修改相应数据。

同时，为了提高利用率，当复合文件有一半以上的空间没有被利用是，进行相应的空间压缩操作。

2.3 复合文件方式

有三种复合文件的方式：

第一种是基于目录的合并，将一个目录中的所有文件组合成一个复合文件，但不包含它本来的子目录。可以在所有目录上执行基于目录的整合，而无需跟踪和分析文件参考。 但是，它不会捕获跨目录的文件关系。

第二种是基于嵌入式参考的合并，根据文件中嵌入的文件引用来合并成复合文件 。例如，超链接可能被嵌入在一个 html 文件中，一个网络爬虫很可能通过这些链接访问每个网页。 在这种情况下，我们合并原始的 html 文件和引用的文件。 这种方法可以识别跨目录访问的相关文件，但是不易提取超出基于文本的文件格式的文件引用。

第三种是基于频率的合并，这里使用了 apriori 算法。利用该算法我们可以得到输入数据的关联性，从而确定谁和谁合并。这里不再多描述该算法。

2.4 性能总结

在 ext4 文件系统上进行实验，可以看到基于嵌入式参考的整合效果最好，高速缓存的请求率为 62％，比 ext4 高 20％。基于目录的复合文件也可以将缓存命中率提高 15％，反映目录捕获空间位置的有效性。基于频率挖掘（frequency-mining-based）的整合表现比基于目录的差，但总体性能都是有提升的。

总得来说，本文所提出的复合文件系统，解除了文件和元数据之间的一对一映射关系。实验结果证明复合文件系统，能够将文件

系统的吞吐量增加 27%，将响应延时减少 20%，表明这种方案具有很大的发展空间，并且可以针对解决文件系统优化问题给予一定的启发。

2.5 相关思考及扩展

将相近的文件放在一起这种思想并不是第一次出现，之前在操作系统中，就有内存管理的方法，比如先进先出，最佳置换，最近最久未使用置换等等。一些性能好一些的算法也是考虑了最近使用的的问题，抽象出来就是考虑了改文件或该页面与其他文件或页面的关联性问题。或许这两个并不甚相同，但在我看来，考虑相关性，让关系比较紧密的文件放在一起处理，这是一个在考虑问题时可以参考的想法，也比较符合我们计算机中的时间相关性、空间相关性。

在现在这个大数据时代，我们的文件系统的元数据需要有新的一步改进，大数据也有其元数据，我觉得这应该也是一个可以研究的方向。企业或团体会有很多的数据，而他们数据的元数据，就不止可以得到权限、时间戳等信息，而可以从这些信息中进一步推断出更多的信息，比如数个文件的关系，从权限级别推断内容的重要性。一方面元数据的管理需要更有效，另一方面也需要更加安全。

第 3 章　尾延迟检测及消除方法

3.1 背景简介

由于为云提供动力的分布式系统已经成熟，一个新的性能重点已经开始发挥作用：尾延迟。长的尾延迟可以极大地损害交互性能，从而限制了可以在现代云服务中有效部署的应用程序。减少尾部延迟的挑战的根本原因是，在大规模系统运行时，对单个系统影响不大的罕见的行为可能占主导地位。 因此，虽然系统的测试良好且频繁执行的部分表现良好，但在一个机器上容易忽略的异常行为变成在一千（或更多）机器上的常见情况。

我们希望能够发现小的但会对大规模运行产生影响的行文，使开发人员能够在部署前找到并修复内部尾部延迟问题。但是这个问题很难解决，时至今日，发现性能差的尾部影响行为仍是一个重大挑战。

3.2 实验设计

尾延迟的一个关键因素是本地文件系统，文中介绍了 Chopper，一个使开发人员能够发现（并随后修复）本地文件系统中的高延迟操作的工具。其中，使用了拉丁超立方体采样和灵敏度分析这样的先进技术到文件系统性能分析的领域，这样做使查找尾部行为易处理。

蒙特卡罗方法是通过对输入重复随机抽样获得数值结果来探索模拟的过程。我们将文件系统本身视为模拟器，从而将其放入蒙特卡罗框架。拉丁超立方体采样（LHS）是一种抽样方法，可有效地探索具有大输入空间的多因素系统，并有助于发现令人惊讶的行为，同时可以有效地发现哪些因素和哪些因素的组合对响应有很大影响。这都是我们会用到的方法。

因为文件系统很复杂，所以实际上不可能研究影响性能的所有可能的因素。根据 Chopper 有三类输入因子：第一类因素描述文件系统的初始状态。第二类包括相关的 OS 状态。第三类包括描述工作负载本身的因素。实验选择 12 个因素（如图 4 所示）来逐一分析。

	Factor	Description	Presented Space
FS	DiskSize	Size of disk the file system is mounted on.	1,2,4,...,64GB
	UsedRatio	Ratio of used disk.	0, 0.2, 0.4, 0.6
	FreeSpaceLayout	Small number indicates high fragmentation.	1,2,...,6
OS	CPUCount	Number of CPUs available.	1,2
Workload	FileSize	Size of file.	8,16,24,...,256KB
	ChunkCount	Number of chunks each file is evenly divided into.	4
	InternalDensity	Degree of sparseness or overwriting.	0.2,0.4,...,2.0
	ChunkOrder	Order of writing the chunks.	permutation(0,1,2,3)
	Fsync	Pattern of `fsync()`.	****, *=0 or 1
	Sync	Pattern of `close()`, `sync()`, and `open()`.	***1, *=0 or 1
	FileCount	Number of files to be written.	1,2
	DirectorySpan	Distance of files in the directory tree.	1,2,3,...,12

图 4.实验的因素

这 12 个因子构成了实验的输入。DiskSize 因子是指创建一个这么多字节的虚拟磁盘，因为块分配器对于不同大小的磁盘可能有不同的空间管理策略。UsedRatio 因子描述已使用的磁盘所占的比率。FreeSpaceLayout 因子描述磁盘上的可用空间的连续性，使用六个数字来表示从极度碎片到一般连续的度数，用自由范围大小的分布来描述碎片的程度，这个范围大小遵循对数正态分布。CPUCount 因子控制操作系统运行的 CPU 数。它可以用于发现块分配器的可扩展性问题。FileSize 因子表示要写入的文件的大小，因为分配器在分配不同大小的文件时可能表现不同。ChunkCount 因子可以捕获块，不同数量的块可以被块分配器使用。InternalDensity 因子描述文件的覆盖

程度，比如说它是稀疏的或者被过度的写。ChunkOrder 因子定义写入块的顺序。 它探索顺序和随机写入模式，但具有更多的控制。Fsync 因子被定义为一个位图，描述在每个块被写入之后 Chopper 是否执行 fsync 调用。Sync 因子定义了如何在每次写入时打开，关闭或同步文件系统。FileCount 因子描述了写入文件的数量，用于探索块分配器如何保留一个文件和多个文件的空间局部。DirectorySpan 因子描述了在树的宽度优先遍历中第一个和最后一个文件的父目录之间的距离。

实验中所使用的 Chopper 工具的多个组件分工协作。Manager 构建了一个实验计划，并使用其他组件执行计划。FS 操作员为后续工作负载准备文件系统。为了加速实验，文件系统安装在内存中的虚拟磁盘上，该磁盘被实现为一个由 RAM 文件系统中的文件支持的环回设备。每当需要时重新使用初始磁盘映像，从而加速实验并提供再现性。映像准备就绪后，工作负载生成器会生成工作负载描述，然后将其导入 Workload Player 以便运行。在播放工作负载后，管理器通知 FS 监视器，该监视器调用现有系统实用程序（如 debugfs 和 xfs_db）来收集布局信息。不需要更改内核。最后，布局信息与工作负载和系统信息合并，并馈入分析器。实验运行可以并行执行以显着减少时间。

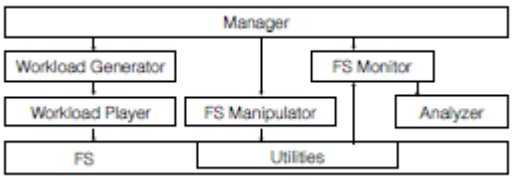

图 5.Chopper 组成

在实验中，为了确定在所得到的分配中是否有尾部，第一，我们使用 Chopper 进行了 16384 次运行的实验。第二，实验在具有 16 GB RAM 和两个 Opteron-242 CPU 的节点集群上进行。第三，利用 Chopper 的并行性和优化，每个文件系统的一个完整实验花了大约 30 分钟与 32 个节点。

3.3 策略改进

在调查工作负载和环境因素中对 ext4 布局中所看到的变化贡献较大的因素，这些因素是重要因素有两个原因，首先，它可以帮助文件系统用户查看哪些工作负载在给定文件系统上运行得最好，并避免那些运行不良的工作负载; 第二，它可以帮助文件系统开发者跟

踪内部策略问题的来源。这里，我们用了敏感性分析技术，就是一个因素对变化的贡献可以通过基于方差的因素优先级计算。

经过实验，我们找到最重要的因素是 DiskSize，FileSize，Sync，ChunkOrder 和 Fsync;也就是说，改变这些因素中的任何一个都可能显着影响 dspan 和布局质量。

在发现 ext4 的布局策略的一些问题后，使用数据分析提供的提示来缩小问题的来源，并且在 Chopper 建议的工作负载集合下执行详细的源代码跟踪达到精确定位，以这种方式来解决 ext4 布局策略中的一系列问题，并显示每个修复减少了 ext4 布局中的尾部情况。

为了无调度程序依赖性，我们通过为一个基于其 inumber 范围而不是 CPU 的小文件选择 locality 组来消除随机布局的问题。使用 i 号码不仅消除了对调度程序的依赖，而且还确保具有接近 i 号码的小文件可能被靠近放置。为了减少布局的可变性，我们从 ext4 中删除了特殊结束策略，这导致 32％的运行的 dspan 减少总共 21TB，但是将 14％的运行的 d 跨度增加总共 9TB。dspan 的增加主要是因为删除此策略会取消隐藏文件大小依赖性中不一致的策略。

但文件系统是复杂的，不同的因素之间也会有互相的影响，删除调度程序依赖性和特殊结束策略后，ext4 布局仍然显示一个显著的尾部。跨工作负载运行的楼梯形状尾部运行表明此策略只影响大文件，它取决于第一个写的块。但是将大文件的范围与共享全局策略一起放置，会违反将大文件分开的初始设计目标，并恶化文件大小依赖性的后果。为了缓解这个问题，实验尝试将大文件的范围放在该文件的现有范围附近。实验结果表明大型文件（> 64KB）的布局得到显着改善，稀疏文件（具有内部密度<1）的布局也得到改进，证明新策略能够单独分配每个范围，同时仍然保持它们彼此靠近。改正以上三个策略后，在 ChunkOrder 和内部密度之间存在有趣的交互，虽然大多数工作负载表现出尾部，但是几个工作负荷没有。通过正确更新请求的物理起始块来修复错误，大文件特别容易受到这个错误的影响，稀疏文件也是如此。

3.4 性能及相关问题

上述四个修正，使得 dspan 值的第 90 百分位显着降低从超过 4GB 到接近 4MB。因此，最终版本的 ext4 比原来的 ext4 有更少的尾部。

虽然这些修复显著减少尾部行为，但它们有几个潜在的限制。首先，没有调度程序依赖性策略，运行在不同 CPU 上的刷新线程可能争用相同的预分配组。默认争用程度是可以接受的，因为预分配内的分配很快，并且文件分布在许多预分配中。如果发现争用是一个问题，可以添加更多的预分配（当前 ext4 延迟创建预分配，每个 CPU 一个预分配）。第二，删除共享全局策略减轻了但不消除具有动态变化大小的文件的布局问题。基于诸如文件大小的动态属性来选择策略是复杂的并且需要更基本的策略修订。第三，修改后的最终版本，仍然包含一个小尾巴。这个尾部来自于于磁盘状态。如所期望的，当文件系统在使用更加频繁且碎片更多的磁盘上运行时，新文件的布局受损。

从实验中，我们也得到了一些今后可以用到的经验来帮助构建出对不确定的工作负载和环境也是健壮的文件系统。首先，针对不同情况的政策应当彼此协调。第二，政策不应取决于可能改变并且不受文件系统控制的环境因素。

3.5 相关思考及扩展

文件系统基准已经被批评了几十年。许多文件系统基准测试目标涉及文件系统性能的许多方面，因此包括许多以不可预测的方式影响结果的因素。 相比之下，Chopper 利用发展良好的统计技术来隔离各种因素的影响，并避免噪音。 Chopper 只专注于块分配，能够隔离其行为并揭示数据布局质量的问题。

有时候对于一个方向或者实验，可能有多个影响因素，如果为了优化或者改进而一股脑的全部考虑，可能会加大实验难度，甚至使实验停滞不前。所以逐个的分析各个因素，找出其中重要的以及相互影响的，然后主义解决逐一分析，从而找到最佳结果。

参考文献

【1】Zhang S, Catanese H, Wang A I A. The Composite-file File System: Decoupling the One-to-One Mapping of Files and Metadata for Better Performance[C]//FAST. 2016: 15-22.

【2】He J, Nguyen D, Arpaci-Dusseau A C, et al. Reducing File System Tail Latencies with Chopper[C]//FAST. 2015, 15: 119-133.

【3】数据仓库与元数据管理。http://blog.csdn.net/zjw00417236/article/details/6120936

【4】百度百科。
https://baike.baidu.com/item/%E5%85%83%E6%95%B0%E6%8D%AE/1946090?fr=aladdin

【5】尾延迟 Tail Latency 。http://blog.csdn.net/guo_jia_liang/article/details/53741775