|  |
| --- |
| **复合文件系统的设计** |

|  |  |
| --- | --- |
| 院（系）名 称 ： | 计算机学院 |
| 专 业 名 称 ： | 计算机软件与理论 |
| 学 生 姓 名 ： | 刘思璇 |
| 学 号 ： | 2017202110027 |

目 录

[1 文件与元数据 1](#_Toc500605122)

[1.1 元数据 1](#_Toc500605123)

[1.2 文件系统中元数据的存储方式 1](#_Toc500605124)

[1.2.1 UNIX文件系统中元数据的存储方式 1](#_Toc500605125)

[1.2.2 并行文件系统元数据的存储方式 2](#_Toc500605126)

[2 复合文件系统的设计 3](#_Toc500605127)

[2.1 传统文件系统 3](#_Toc500605128)

[2.1.1 小型文件的频繁访问 3](#_Toc500605129)

[2.1.2 元数据的信息冗余 3](#_Toc500605130)

[2.1.3 成组访问文件的缺陷 3](#_Toc500605131)

[2.1.4 预取文件时的限制 3](#_Toc500605132)

[2.2 设计概述 4](#_Toc500605133)

[2.3 具体数据处理 4](#_Toc500605134)

[2.3.1 复合文件的创建 4](#_Toc500605135)

[2.3.2 节点创建 5](#_Toc500605136)

[2.3.3 权限设定 5](#_Toc500605137)

[2.3.4 时间戳 5](#_Toc500605138)

[2.3.5 文件大小 6](#_Toc500605139)

[2.3.6 子文件的设定 6](#_Toc500605140)

[2.4 文件合并方法 6](#_Toc500605141)

[2.4.1 基于目录的文件合并方法 6](#_Toc500605142)

[2.4.2 基于嵌入式参考的文件合并方法 6](#_Toc500605143)

[2.4.3 基于访问频率的文件合并方法 7](#_Toc500605144)

[3 复合文件系统的实现与评估 9](#_Toc500605145)

[3.1 复合文件系统的实现 9](#_Toc500605146)

[3.2 复合文件系统的评估 9](#_Toc500605147)

[参考文献 12](#_Toc500605148)

# 文件与元数据

文件系统是用来管理组织数据的方法，而I/O密集型应用使I/O子系统成为整个计算机系统的瓶颈，文件系统作为底层存储资源的管理者，应该为上层应用提供灵活有效的服务，以更有效地利用底层存储资源。与此同时，文件系统中元数据的管理对于很多数据密集型程序也成为了瓶颈。

## 元数据

数据，又称中介数据、中继数据，为描述数据的数据，主要是描述数据属性的信息，用来支持如指示存储位置、历史数据、资源查找、文件记录等功能。元数据算是一种电子式目录，为了达到编制目录的目的，必须在描述并收藏数据的内容或特色，进而达成协助数据检索的目的

元数据就是数据的数据。任何文件系统中的数据分为数据和元数据。数据是指普通文件中的实际数据，而元数据指用来描述一个文件的特征的系统数据，诸如访问权限、文件拥有者以及文件数据块的分布信息(inode...)等等。在集群文件系统中，分布信息包括文件在磁盘上的位置以及磁盘在集群中的位置。用户需要操作一个文件必须首先得到它的元数据，才能定位到文件的位置并且得到文件的内容或相关属性。

文件系统元数据的更改都被保存在一份单独的日志里，当发生系统崩溃时可以根据日志正确地恢复数据。除此之外，日志使系统重新启动时不必进行文件系统的检查，从而缩短了恢复时间。

## 文件系统中元数据的存储方式

### UNIX文件系统中元数据的存储方式

UNIX文件系统中存储元数据的结构叫做inode，inode结构中存储了文件系统对象（包括文件，目录，设备节点，套接字，管道等）除了数据内容和文件名字以外的所有信息[1]。 inode结构中不包括文件名字，而只包含文件的元数据。UNIX的目录文件则是一个列表，每个列表项包括一个文件或者目录的名字，以及其相对应的inode 编号，因此文件系统必须查找目录文件才能将文件的名字转换成其对应的inode编号。每个文件都对应一个唯一的inode节点，而每个inode节点却对应了不止一个文件，主要是因为UNIX文件系统中存在着硬链接（hard link）。 上面包括模式、文件拥有者、时间戳、块大小、计数器还有指向文件数据块的指针。对于文件数据的存储位置在inode节点体现为四个级的别指针。inode节点中包括12个直接指针，1个一重间接指针，1个两重间接指针，1个三重间接指针。直接指针可以直接指向文件数据的块号，而一重间接指针则指向一个间接块，而这个间接块上包括多个可以指向数据块的指针，同理对于两重间接指针，三重间接指针则是中间通过两重或三重的间接块，然后再指向具体的文件数据块。

### 并行文件系统元数据的存储方式

单个I/O子系统性能的提高非常缓慢，并行文件系统通过并行化I/O操作提供更高的I/O性能。并行文件系统一般包括数据服务器和元数据服务器，提供一个全局的命名空间，可扩展性，以及将大文件分配到多个节点上的能力。元数据服务器上包括数据服务器上文件的信息，比如文件名字、位置、所有者等等。一些并行文件系统使用专门的服务器作为元数据服务器，有一些并行文件系统利用I/O节点同时作为元数据服务器，提供元数据访问。 将元数据与数据分开存储从而可以将元数据负载和数据的负载分开，对于提高整体的I/O性能有很大的意义。同时如果存在多个元数据服务器，还可以通过不同的负载均衡技术，平衡不同元数据服务器的负载，从而提高系统整体的吞吐率；进一步来讲，多个元数据单个I/O子系统性能的提高非常缓慢，并行文件系统通过并行化I/O操作提供更高的I/O性能。并行文件系统一般包括数据服务器和元数据服务器，提供一个全局的命名空间，可扩展性，以及将大文件分配到多个节点上的能力。元数据服务器上包括数据服务器上文件的信息，比如文件名字、位置、所有者等等。一些并行文件系统使用专门的服务器作为元数据服务器，有一些并行文件系统利用I/O节点同时作为元数据服务器，提供元数据访问。 将元数据与数据分开存储从而可以将元数据负载和数据的负载分开，对于提高整体的I/O性能有很大的意义。同时如果存在多个元数据服务器，还可以通过不同的负载均衡技术，平衡不同元数据服务器的负载，从而提高系统整体的吞吐率；进一步来讲，多个元数据

# 复合文件系统的设计

## 传统文件系统

### 小型文件的频繁访问

文件目录的大部分其实都是小型文件[2]，据统计在我们对桌面文件进行访问时，超过百分之八十访问的都是小于32字节的小型文件，而访问磁盘上的小文件的访问时间的40%可以归因于元数据访问，所以减少访问时间可以很大程度上的获得性能增益。

### 元数据的信息冗余

传统文件与自己的物理元数据是相互关联的，元数据记录了文件的位置、访问权限等信息，但是由于文件所有者、权限模式等属性的数量是有限的，所以有许多文件可以共享类似的文件属性，Edel等人的研究[3]表明对于传统的工作站来说，元数据的压缩率可以达到约百分之七十五，所以，对于减少元数据的信息冗余有很大的进步空间。

### 成组访问文件的缺陷

文件在被访问时一般是进行成组访问的[4][5][6]，例如在进行Web访问时，通常还会涉及到访问许多相关文件，考虑到识别和分组文件的过程会导致开销，所以利用文件分组的优化可能不会产生自动性能增益。

### 预取文件时的限制

预取是一种很有效的优化方式，但是单独提取每个文件和访问相关元数据的行为可能会导致高开销，例如在单独访问32个小文件时的总延迟可能比访问一个大小等于32个文件之和的单独文件，甚至是通过热缓存，延迟也高了将近百分之五十。

## 设计概述

复合文件系统（CFFS）介绍了一种通过保存经常访问的小文件的内部物理访问形式。对于终端用户来说，可以通过小文件之间共享的i-node来对复合文件进行访问，存储在这些节点中的文件原始信息中重复的部分会被删除，并作为复合文件的扩展属性存储。每一个单独小文件的元数据属性仍然可以进行重建、检查、更新的操作，所以传统的访问方式不会改变，而扩展属性还记录了文件中单个小文件的位置，因此我们可以将一个物理复合文件转变成逻辑文件。

## 具体数据处理

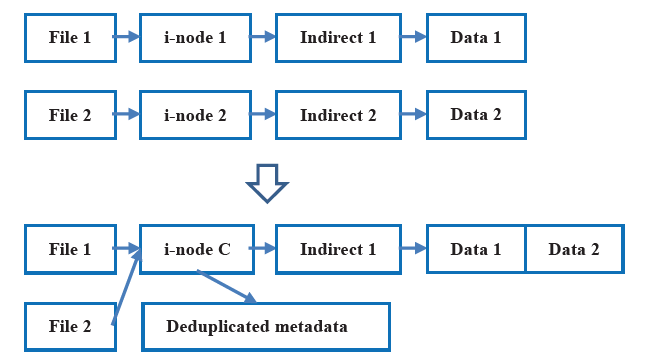
一个复合文件是由很多个小文件组合而成的，这些小文件称为子文件。一个复合文件中所有的子文件、间接块、双间接块共享同一个i-node。由于是由多个小文件组合而成，复合文件不用考虑文件占用空间的最大限制，因为即使子文件的大小总和超过了这个限制，我们也可以通过采取分成多个复合文件来解决。

通常来说，每个复合文件的第一个子文件都是入口点，通过访问这个子文件可以使剩余的子文件进行预取操作。举例来说，当浏览器访问一个HTML文件时，它需要加载一个CSS文件和一个Flash脚本，这个HTML文件可以作为这三个子文件组成的复合文件的入口点并触发另外两个文件的预取操作。由于有多个子文件，所以当这些子文件以不同顺序排列时，这个复合文件会有不同的入口点，这个时候我们通常通过文件访问频率来对数据进行布局。

### 复合文件的创建

当我们创建一个复合文件时，系统会分配给这个文件一个i-node并且会连接复制其所有子文件的内容作为这个复合文件的数据。该文件同时会记录复合文件的偏移量以及每个子文件的大小，并且将已经删除了重复数据的i-node信息作为扩展属性存储。之后，原来的子文件会被截断，同时它们的目录根据新的i-node节点和子文件的ID重新进行映射，释放原节点信息。图2.1所示为由两个原始文件创建一个内部复合文件的过程，每个子文件能被同样的方式找到，终端用户仍然认为这些单独的逻辑文件在同样的位置存储。

图2.1 复合文件创建示意图



### 节点创建

删除了重复数据的子文件节点在重建时的速度是很快的，在默认情况下，除非有特别指定的扩展属性，否则子文件的i-node会继承复合文件的i-node值。当i-node的数量大于我们设定的阈值时，上零扩展N位用于复合节点数，低M位用于子文件的ID，我们把这个作为CFFS的独特IDs（CUIDs）。

### 权限设定

在打开文件时，首先会对复合节点进行权限检测，如果没有权限，则不需要继续检测，否则，当其中一个子文件具有不同的权限作文扩展属性存储时，会继续进行权限检测，所以这个复合节点将会拥有所有子文件的广泛权限。举例来说，在一个复合文件内，我们有一个只读的子文件A和一个可写的子文件B，复合节点的权限将会是可读可写，然而，当我们用写操作打开子文件A时，它本身存储在扩展属性的只读权限会捕捉到这个异常。

### 时间戳

每一个子文件和复合文件的时间戳都会在每次进行文件操作后更新，但是在检查时，如进行stat ( )系统调用时，我们会返回子文件的时间戳。

### 文件大小

复合文件的大小可以大于它所有子文件大小的总和，举例来说，如果一个位于复合文件中部的子文件被删除了，它的空间被释放，但是复合文件的大小并没有改变。

### 子文件的设定

如果目录中的一个文件名映射到CUID，则一个子文件的属性可以通过子文件的ID查找到。由于从一个复合文件中添加或删除一个子文件会改变它的CUID，所以我们需要存储后指针[7]用于更新所有映射的CUID。当我们把一个子文件添加到复合文件中时，它会添加到复合文件的末尾，当我们把一个子文件从复合文件中删除时，复合文件中相应的数据区域内的扩展属性会被释放。

当我们对子文件进行打开或者关闭操作时，随着文件位置指针的改变，我们同样可以看做是在对这个复合文件进行打开或关闭操作。对于更新操作时，复合文件的处理方式与传统文件系统是相同的，但是当更新涉及到在复合文件中部生成子文件且在子文件的末尾没有可用空间时，则将更新的子文件移动到复合文件的末尾。该方案利用了潜在的时间局部性，即不断增长的子文件可能在不久之后再次增长。

## 文件合并方法

### 基于目录的文件合并方法

目录是文件访问模式和形成复合文件的一个很好的模拟，目前，这个合并计划还没有包括子目录。基于目录的合并文件方法无需跟踪和分析文件目录，但是，它不会捕获跨目录的文件关系，

### 基于嵌入式参考的文件合并方法

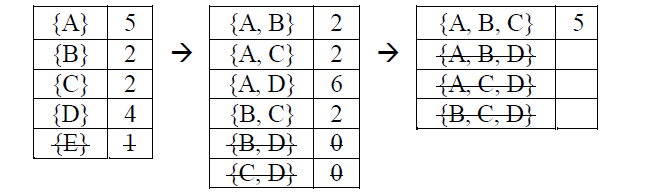
基于嵌入式参考的合并根据文件中嵌入的文件引用来标识复合文件成员资格。 例如，超链接可能被嵌入在一个html文件中，一个网络爬虫很可能通过这些链接访问每个网页。在这种情况下，我们可以合并原始的html文件和引用的文件。类似的想法同样适用于编译。我们可以从Makefile中提取依赖关系规则，并合并可以生成相同二进制文件的源文件。由于文件更新可能会打破依赖关系，CFFS可以通过定期修改的文件进行筛选以协调复合文件成员资格。

基于嵌入式参考的方案可以识别跨目录访问的相关文件，但是可能不容易提取超出基于文本的文件格式（例如，HTML，源代码）的嵌入式文件引用，这样做的前提是它需要知道特定的文件格式。

### 基于访问频率的文件合并方法

在对基于访问频率的文件合并方法的研究中，使用了Apriori算法[8]的一个变种。 由Apriori属性可知，如果一组文件为频繁集，则它的任意子集也是频繁集。 图2.2为对文件A，B，C，D和E的访问流的算法流程。

图2.2 Apriori算法流程



首先，我们计算每个文件的访问次数，然后删除计数小于阈值（如两个）的文件以供进一步分析。对于其余的文件，我们排列，构建和计数所有可能的双文件引用集。只要在B之后立即访问文件A，反之亦然，我们增加文件集{A，B}的计数。计数小于阈值的集被删除（例如，{B，D}）。我们可以根据剩余的两个文件参考集合生成所有三个文件的参考集。但是，如果一个三文件参考集经常出现，那么它所有的两文件参考集也需要频繁出现。因此，{A，B，D}等文件集被修剪，因为它的子集{B，D}在第二步中已经被去除了。最后，由于我们不能再生成四个文件的参考集，算法结束。现在，如果一个文件可以属于多个文件集，我们将集合{A，B，C}和{A，D}作为两个经常访问的集合返回。诸如{A，B}的集合被删除，因为它们是{A，B，C}的子集。

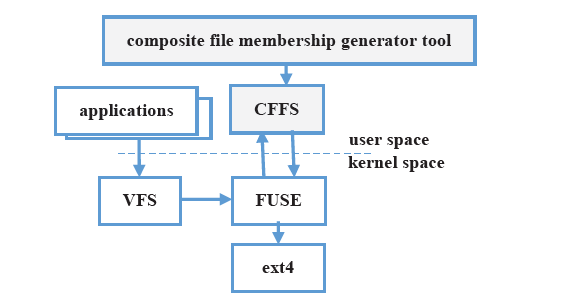
另一种方法是使用标准化的阈值或支持度，即设置事件的百分比（一组事件的发生次数除以总事件的次数，范围介于0和1之间）。我们也可以跟踪文件参考序列来确定合成文件的入口点和内容布局，而不是跟踪文件集。目前还不允许出现重叠的文件集，以避免因重复而产生的复杂性。在两个复合文件之间选择子文件成员资格，取决于复合文件是否有更多的子文件，更高的支持和更新的创建时间戳。基于访问频率的文件合并方法可以基于动态文件引用来识别复合文件候选。

# 复合文件系统的实现与评估

## 复合文件系统的实现

该系统通过在Linux 3.16.7上运行如图3.1所示的FUSE（v2.9.3）框架在用户空间中创建了CFFS原型。 CFFS堆叠在ext4上，以便利用旧版工具和功能，如持久性引导（例如文件系统创建实用程序），扩展属性和日志记录。

图3.1 CFFS组件（阴影）和从应用程序到底层ext4的数据路径



CFFS定期与生成器工具进行协商以创建新的复合文件。系统通过利用类似于硬链接的机制来允许将多个文件名映射到相同的复合i节点。由于需要更新各个子文件的时间戳，需要拦截所有与文件系统相关的调用。同时需要确保各种访问使用正确的权限（例如open和readdir），已翻译的子文件偏移和大小（例如，读和写）以及时间戳（例如，getattr和setattr）。实际的组合文件，其i节点及其扩展属性由底层的ext4文件系统存储。

## 复合文件系统的评估

图3.2显示了磁盘的Web服务器请求延迟的CDF，从请求发送到请求完成的总时间。系统设计的最初意图是减少经常访问的小文件的IO数量。 但是，对统一元数据访问次数较少的好处是将自身显示为所有子文件的元数据预取，复合文件语义启用跨文件预取，从而实现更高的高速缓存命中率。从图中可知，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%。 因此，基于嵌入式引用创建的复合文件更准确地捕获访问模式。 整体重播时间也减少了20%左右。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，但是基于访问频率的整合表现比基于目录的更差。

图3.2 HDD的Web服务器请求延迟

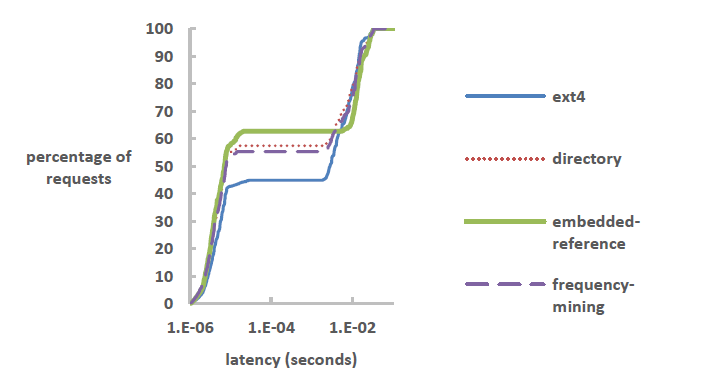
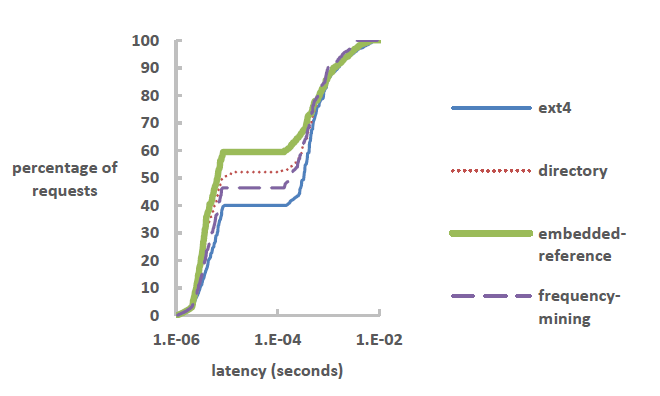


图3.2显示了SSD的Web服务器请求延迟的CDF。与磁盘相比，大致的趋势相似，但是由于SSD的速度更快，缓存未命中的请求延迟时间减少了两个数量级。 由于主要的性能提升是由较高的缓存命中率和IO回避造成的，因此这20％的收益与底层存储介质相当独立。

图3.3 SSD的Web服务器请求延迟



参考文献

[1] WEIL, S. A., BRANDT, S. A., MILLER, E. L., LONG, D. D. E., AND MALTZAHN, C. Ceph: A scalable, high-performance distributed file system. In Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI) (2006), pp. 307–320.

[2] Roselli D, Lorch JR, Anderson TE. A Comparison of File System Workloads. Proceeding of 2000 USENIX Annual Technical Conference (ATC), 2000.

[3] Edel NK, Tuteja D, Miller EL, Brandt SA. Proceedings of the IEEE Computer Society’s 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, 2004.

[4] Kroeger TM, Long DE. Design and Implementation of a Predictive File Prefetching. Proceedings of the USENIX 2001 Annual Technical Conference (ATC), 2001.

[5] Li Z, Chen Z, Srinivasan SM, Zhou YY. C-Miner: Mining Block Correlations in Storage Systems. Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST), 2004.

[6] Ding X, Jiang S, Chen F, Davis K, Zhang X. DiskSeen: Exploiting Disk Layout and Access History to Enhance Prefetch. Proceedings of the2007 USENIX Annual Technical Conference (ATC), 2007.

[7] Chidambaram V, Sharma T, Arpaci-Dusseau AC, Arpaci-Dusseau RH. Consistency without Ordering. Proceedings of the 10th USENIX Conference on File and Storage Technologies, 2012.

[8] Agrawal R, Srikant R. Fast Algorithms for Mining Association Rules, Proceedings of the 20th VLDB Conference, 1994