武汉大学计算机学院

|  |
| --- |
| 姓名：柳登凯 |
| 学号：2017202110095 |
| 学院：计算机学院 |
| 专业：信息安全 |
| 论文：The Composite-file File System: Decoupling the One-to-one Mapping of Files and Metadata for Better Performance |

2017/12/8

composite - File文件系统:去除小文件和元数据的一对一映射

**简介:** 传统的文件系统优化通常使用逻辑文件的到物理元数据一对一映射。这种映射会大大增加存储空间代价，且相比较于去除掉这种一对一映射的文件系统，传统文件系统失去了很多优化的机会。这篇论文设计、实现和评估了一个composite - file文件系统，它允许多个文件映射到同一个元数据节点，同时设计评估了不同种映射策略的优劣效果。在web服务器和软件开发工作负载下，实验结果显示性能提高了27%。这个结果表明这种复合文件具有广阔的应用前景。

# 1 简介

文件系统的性能优化是一个热门领域。然而，大多数优化技术(例如:缓存，更好的数据布局)保留逻辑文件的一对一映射到它们的物理元数据表示(即，每个文件都与UNIX平台上自己的i节点相关联)。这种映射是可取的，因为元数据构造是最基本的数据结构，许多存储组件和机制(比如VFS API[MCK90]、预取和元数据缓存)依赖于这些构造。然而，这种刚性映射也为很多的性能优化提供了一个盲点。

本篇论文设计、实现和评估了一个composite - file文件系统（CFFS），许多逻辑文件可以组合在一起，并与单个i节点(加上存储为扩展属性的额外信息)相关联。这样的安排是可能的，因为许多文件共享相同的元数据子字段[EDL04]，可以被删除而不需要占用额外的存储空间。因此，CFFS可以减少对存储的元数据访问，这是访问小文件的主要开销的来源，而这些文件仍然在大多数现代工作负载的文件引用中占主导地位[ROS00;HAR11]。

基于web服务器和软件开发工作负载，CFFS可以超过ext4，达到27%，这表明去掉文件到元数据映射的方法是很有前景的。

# 2 背景和意义

在以往文件系统的优化过程中，有以下若干现象或者说是传统文件系统的去诶单和不足之处。他们分别是小文件的频繁访问，数据信息一般存在冗余，文件往往按组访问，文件预取机制等等。

而以下这些观察结果正是CFFS的设计灵感来源:

*小文件的频繁访问：*研究[ROS00;HAR11]大部分的文件引用或者说访问目标是小文件。通过对桌面文件系统的分析证实了大于80%的访问都是小于32字节的文件。此外，访问磁盘上的一个小文件的访问时间的40%可以归因于元数据访问。也就是说对于传统的一对一元数据映射过程来说，减少这种访问开销可能会带来很大的性能收益。

*冗余元数据信息:*传统的文件与它自己的物理元数据相关联，它记录下了文件块位置、访问权限等信息。然而，许多文件共享类似的文件属性，因为文件所有者的数量、权限模式等都是有限的，所以这些共享信息集成到一个元数据中是可能的。Edel等[2004]显示了典型工作站的75%元数据压缩比，意思是说有3/4的元数据是可以被共享的，并非不可替代。因此，我们看到了许多减少冗余元数据信息的机会。

*按组访问的文件*:文件倾向于一起访问，如[KRO01,LI04,DIN07，和JIA13]，这体现了文件访问时的相关性。例如，web访问通常涉及访问许多相关的文件。然而，由于识别和分组文件的过程会带来开销，因此利用文件分组的优化可能不会产生自动性能增益，反之，如果这些相关文件可以映射到一个元数据中，可以得到性能的提升。

*预抓取的限制:*尽管预抓取是一种有效的优化，这也是很多传统文件系统优化常用的策略，但同样的获取每个文件和相关元数据访问的单独行为可以带来很高的开销，因为文件与元数据一一对应，这在预抓取时是相当不利的。

例如，访问32个小文件的延迟比访问单个文件的延迟要高50%，这个文件的大小等于32个文件的总和，即使是热缓存。

以上观察结果引出了一个问题，我们是否可以通过整合访问的小文件来提高性能。这是通过将逻辑文件的一对一映射与元数据的物理表示分离的方法实现的。

# 3 复合文件文件系统

在以上讨论的基础上，本篇论文提出了CFFS系统，它可以整合若干小文件并分享一个i-node，以达到性能提升的效果。该章将首先介绍CFFS的整体设计原理，然后从复合文件的数据表示、元数据的表示和基本操作以及如何整合复合文件成员三个方面进行论述，展示CFFS系统的设计过程。

## 3.1 设计总览

CFFS引入了一个称为复合文件的内部物理表示，它包含经常被访问的小文件的内容，并且共享一个i-node。一个复合文件对终端用户透明，且与一个单独的为小文件所共享的复合i节点相关联。存储在小文件的i节点中的原始信息被复制并存储为复合文件的扩展属性。单个小文件的元数据属性仍然可以存储、检查和更新，以便之前的访问语义(例如:类型、权限、时间戳)都不会被改变。扩展属性还可以记录单个小文件在复合文件中的位置，以便于检索访问。通过这种表示，CFFS可以将物理复合文件转换成逻辑文件，意思是小文件的物理存储位置并没有发生变化，但是这些小文件共享一个i-node共同组成一个复合文件，再被访问时看做是一个整体，通过偏移量来确定某一个小文件，及逻辑上是一个整体，但物理上由若干个小文件组成。

但是这里又引入了一个新的问题：将哪些文件合并到组合文件中？这是一个重要的依赖于工作负载的策略决策方案。在本篇论文中，CFFS配置了三种方式：第一个方案是基于目录的整合，顾名思义，同一目录中的所有文件(不包括子目录)构成一个复合文件；第二种方案是嵌入式引用整合，在文件内容中提取文件引用来识别可以构成复合文件的文件，可以理解为基于内容的整合，例如一些html文件里面嵌入的各种链接所扩展的内容被划分到同一个符合文件中；第三种是基于频率-挖掘的整合，这里会用到数据挖掘中的Apriori算法，类似于求解频繁项集，通过计算文件访问频率来分析文件在一段时间内被访问的频繁程度，因此经常一起访问的文件组成复合文件，这也是基于一个经验：近期内被访问的文件很有可能在将来再次被访问。

复合文件使用传统的VFS预取机制，因为可以用与FFS将小数据块组合成少量的大数据块（可以有效提升性能）相类似的方式将整个复合文件作为单元进行预取，同样有利于性能优化。

## 3.2 数据表示

正如前面所讨论的，复合文件的内容由若干小文件(称为子文件) 根据不同的整合策略整合而成。复合文件中的所有子文件共享相同的i -节点，以及间接块，双重间接块等。复合文件的最大大小限制不是问题，因为复合文件被设计成组合若干个子文件，如果子文件大小的总和超过最大文件大小限制，那就可以分成两个或者多个复合文件来表示。

通常，复合文件中的第一个子文件是入口点，它的访问将触发剩余子文件的预取。举个例子，浏览器如访问一个html文件，它会同时加载css文件和flash脚本。html文件可以作为这个三个子文件复合文件的入口点和预取触发器。对于基于访问频率的整合，子文件的顺序反映了它们是如何被访问的。尽管同一组文件可能具有不同的入口点，但数据布局是基于目前主流的访问模式，并不会与现有的访问模式发生冲突。

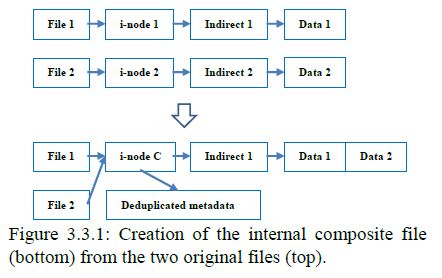
## 3.3 元数据表示和操作

实际上，CFFS文件系统的关键就是元数据的基本表示和可能的操作，其内容包括符合文件的创建，i-node内容重构，文件访问权限，时间戳，i-node命名空间和子文件检索和重命名，子文件成员更新，子文件打开与关闭，子文件写操作，硬链接，空间压缩，子文件并发更新等，以下将逐一阐释说明。

**复合文件创建:**当创建复合文件时，CFFS会分配一个i -node，并将子文件的内容作为其数据复制和连接。复合文件记录了单个子文件的复合文件偏移量和大小，以及它们在扩展属性中删除的i -node信息，然后，原始的子文件被截断，它们的目录条目被重新映射到复合文件的i –node中，扩展属性还包括子文件ID和它们的原始的被删除的i节点。因此，终端用户仍然可以在名称空间中感知单个的逻辑文件，而单个子文件仍然可以定位(图3.3.1，figure 3.3.1)。

**i节点内容重建:**立即删除被复制的子文件i-node。也就是说，一旦子文件信息被复制进复合文件中，原有的子文件i-node将立刻被删除。默认情况下，子文件的i - node字段继承了复合文件i - node字段的值，除非扩展属性中特别说明。

**权限：**当文件打开时，首先根据复合i节点进行权限检查。如果访问成功，不再需要额外的检查，但是如果访问失败，即子文件拥有自己的不同于复合文件的权限作为扩展属性，则需要重新检查。因此，复合i节点将具有跨所有子文件的最广泛的权限。例如，如果在一个复合文件中，我们有一个只读的子文件A和一个可写的子文件B，那么该复合i节点的权限将是读写。然而，当打开一个带有写权限的子文件A时，扩展属性中的只读权限限制将会阻止该次访问。



**时间戳：**个体子文件和复合文件的时间戳在每次文件操作后更新，但是，在检查期间（例如，stat系统调用），返回子文件的时间戳。

**大小：**对于数据访问，偏移量是通过在扩展属性中编码的子文件偏移量和大小来确定的，并且要对偏移量来进行边界检查。一个复合文件的的大小是指这个复合文件的长度，它有可能比它的子文件加起来的总长度还要长。因为如果一个位于复合文件中间的子文件被删除，该空间被释放，却不会改变该复合文件的大小。

**i-node命名空间：**如果一个i-node的编号大于某个阈值X，零扩展高N位就可以用来表示这个i-node，而低M位则被保留用来表示子文件，本篇论文中将i-node编号的这种变化范围叫做CFFS unique IDs（CUIDs）。

**子文件检索和重命名：**如果一个目录名被映射到一个CUID，那么该目录下的子文件的属性可以通过子文件ID来获取。重命名则会和非CFFS系统中一样，一个CUID会当做i-node编号。因为在复合文件中加入和删除一个子文件都会改变它的CUID，所以我们需要存储反向指针用来更新所有映射到这个CUID的文件名。

对CUID的改变可能会破坏一些仅靠i-node编号来识别文件的应用（如备份）。然而，现在的文件系统中，不同的文件也可以在不同的时间片共享同一个i-node编号；CFFS的设计进一步强调了应用不应该把一个i-node编号当做一个文件的私有财产。

**子文件和子文件成员更新：**当一个子文件被添加到一个复合文件中时，它被追加到复合文件尾部；当一个子文件从复合文件中删除时，在该复合文件扩展属性中相应数据空间被标记为已释放。

**子文件打开/关闭操作：**对一个子文件的打开/关闭调用与对复合文件的打开/关闭调用一样，文件位置指针将被读取。

**子文件写操作：**当前位置更新的处理方式与传统文件系统一致，然而，如果更新发生在一个复合文件的中间区域且涉及子文件的增长并且子文件尾部没有多余的空闲空间，就会把该更新子文件移到复合文件尾部。这种机制利用了潜在的时间局部性：一个增长中的子文件很有可能在不久的将来再次增长。

**硬链接：**字典中的不同名字可以被映射到同一个i-node编号或者CUID。

**空间压缩：**当半数的分配空间不包含有用的数据时，复合文件会压缩它的空间。

**复合文件内的子文件并发更新：**在复合文件中对子文件的并发更新与对普通文件的并发更新具有相同的语义。为了避免锁争用，检测到要参与并发更新的文件可能需要被提取到多个常规文件中。

## 3.4 识别复合文件成员

### 3.4.1 基于目录的整合

由于遗留文件系统的空间局部性优化是围绕目录进行的，因此目录是文件访问模式和形成复合文件的很好的近似值，也就是说，对同一个目录下的子文件添加进同一个复合文件。但是目前为止，这个整合方案不包括子目录。

可以在所有目录上执行基于目录的整合，而不需要跟踪和分析文件引用。但是，它不会捕获跨目录的文件关系，但是在很多情况下，有相关关系的文件位于不同的目录下。

### 3.4.2 基于嵌入引用的整合

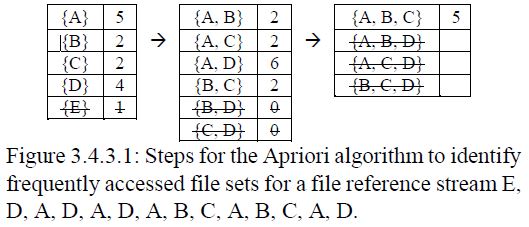
基于嵌入引用的整合是基于文件中的嵌入式文件引用来识别复合文件成员的。例如，超链接可以嵌入到一个html文件中，而网络爬虫很可能通过这些链接访问每个网页。举个例子，分析时可以把原始的html文件和引用的文件一起划入同一个复合文件。类似的想法也适用于编译，首先从Makefile中提取依赖规则，并把会生成相同二进制文件的源文件合并到一个复合文件中。由于文件更新可能会破坏依赖关系，CFFS可以通过修改后的文件周期性地进行筛选，以协调复合文件的成员关系。

基于嵌入引用的方案可以识别跨目录访问的相关文件，但是从基于文本的文件格式(例如html、源代码)中提取嵌入式文件引用可能并不容易，也就是说，可能付出的时间代价比较大。此外，它还需要了解特定的文件格式，因此在处理上也并不容易。

### 3.4.3 基于频率挖掘的整合

对于基于频率整合的探索，作者使用了Apriori算法的一个变体[AGR94]。我们知道在数据挖掘中，对于一组事件集{}，其中每个事件包含若干个操作{,, … ,}，统计两个或者多个操作在同一事件中出现的次数，设定一个阈值称为支持度，若出现次数除以事件总数超过支持度，则称这两个或多个操作为频繁项，即很有可能这些操作同时出现，这些频繁项的集合称为频繁项集。Apriori算法关键的思想是从单个操作开始搜索单一元素频繁项，进而迭代寻找二元乃至多元频繁项，组成频繁项集，也就是说多元频繁项的子集一定也是频繁项。在这里基本思想类似，如果一组文件被经常访问，它的子文件集也会被频繁访问(Apriori property)，那么就可以考虑把这些很有可能同时被访问的子文件编入一个复合文件中。图3.4.3.1演示了Apriori算法在寻找文件A、B、C、D和E中会被同时访问的频繁文件集的算法。

**第一轮：**首先，我们计算每个文件被访问的次数，然后去掉访问次数低于某个阈值的文件以便接下来的分析，例如文件{E}。



**第二轮：**对于剩下来的文件，对它们进行组合，并且对所有的二文件组访问次数计数，举个例子，二文件组中的两个文件只要同时被访问，不管A在B后被访问，还是B在A后被访问，文件组{A,B}访问次数加一。同理，访问次数低于某一阈值（例子中阈值为2）的文件组被去掉，不参与接下来的讨论（例如文件组{B,D}，{C,D}）。

**第三轮：**基于第二轮的结果也就是所有的剩下的二文件引用组，生成所有的三文件引用组，同样计算三文件组出现的次数。因为如果三文件引用组频繁被访问，那么它的二文件组子集也必须是频繁被访问，所以文件组例如{A,B,D}被去除，因为它的子集{B,D}在第二轮中被去掉了。

**终止：**当不存在四文件引用组时，算法终止。最后，算法返回文件组{A,B,C}和{A,D}是被频繁访问的文件组，如{A,B}和{A,C}因为是{A,B,C}的子集，所以被去除以避免重复。

**变化：**另一种方法是使用规范化的阈值，或者支持度，即子文件集被访问的百分比(即一个子文件集的被访问次数除以所有的文件被访问次数，取值范围从0到1)。

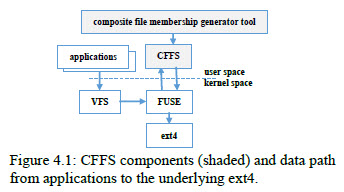
复合文件系统还可以分析文件引用时间序列来确定复合文件的入口点和内容布局，也就是说根据同时出现文件的不同引用顺序来确定复合文件入口点以集资文件排列顺序。但是目前复合文件系统不允许重叠的文件集，以避免复制的复杂性和保持一致性。意思是如果一个子文件有可能属于两个复合文件系统，为了避免出现上面的问题，必须从中选择一个复合文件，这取决于两者中哪个拥有更多的子文件，更高的支持度以及创建时间戳更近。

基于频率挖掘的整合依赖于动态的文件引用也就是文件访问频率来识别整合它的成员，可以达到较好的效果，但是这种整合方法的时间代价很大，也就意味着将它应用到那些被频繁访问的文件序列中受到很大的限制，因为动态更新所要付出的时间代价相当大。

# 4.实现

该篇论文中所提出的新型文件系统主要由两个部分组成：复合文件成员生成器以及CFFS。在图4.1(Figure 4.1)中，是作者通过FUSE(v2.9.3)框架[SZE05]在Linux 3.16.7上运行了用户空间中的CFFS。CFFS在ext4上堆积，因此可以利用现有的工具和特性，比如持久性bootstrapping(例如，文件系统创建工具)、扩展属性和日志记录。

CFFS定期与生成器工具沟通协调，以确定是否需要创建新的复合文件。利用类似于硬链接的机制，允许将多个文件名映射到相同的复合i-node。由于需要更新单个子文件的时间戳，在这个过程中截获了所有与文件系统相关的调用。与此同时还需要确保不同的访问使用正确的权限(例如，open和readdir)，读取子文件偏移量和大小(例如，读写)，以及时间戳(例如，getattr和setattr)。实际的复合文件、它的i-node及其扩展属性存储在底层ext4文件系统中。CFFS可以利用c++实现，大概有1600个分号，也就是大概1600行代码。



对于基于目录的整合，作者使用了一个Perl脚本将目录中的所有文件列出为复合文件成员。对于嵌入式基于引用的方案，主要有两种情况。对于web服务器工作负载，实践中可以合并html文件和它们内容中的引用文件，在复合文件成员冲突的情况下，优先考虑index.html，然后是第一个包含一个文件的html文件。另一个是源代码编译。我们使用Makefile作为一个指南来整合源代码文件，也就是前面所提到的把会生成相同二进制文件的源文件合并到一个复合文件中。对于基于频率挖掘的方案，成员生成器工具利用http访问日志或strace输出。生成器实现了Apriori算法，支持度被设置为5%。每一批引用文件数为50k个。根据经验选择参数，以限制内存和处理开销。生成器代码包含大概有1200个分号，也就是大概有1200行代码。

# 5.性能评估

我们通过将基准ext4文件系统(通过一个空的FUSE模块路由的请求)与对ext4上构建CFFS的性能进行了比较。

我们通过检测两个实时追踪结果来评估我们的系统。第一个是来自一个web服务器的http日志(01/01/2015-03/18 / 2015)。该跟踪包含1400万个文件引用到1.0 tb的数据;其中，有310万个文件是唯一的，持有76GB的数据。第二个跟踪是通过strace从一个软件开发工作站收集的(11/20/2014 - 11/30/2014)。该跟踪包含超过2.4亿的文件系统相关系统调用的24GB数据;其中，291133个文件是唯一的，有2.9 gb字节。读和写操作之间，59%是读，41%是写。以上这些数据表明，绝大部分文件是复用的，也即是只有很少的文件是唯一性的，并且涉及的数据量也较小。

作者在存储设备上执行多线程、零思考时间的跟踪，而且跳过了没有活动的跟踪间隔。重播实验在戴尔工作站上进行，表5.1是工作站的具体配置信息。每个实验重复5次，结果以90%置信区间显示。

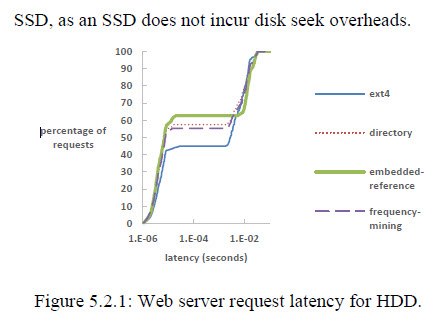
Table 5.1:Experimental platform.

|  |  |
| --- | --- |
| Processor | 2.8GHz Intel® Xeon® E5-1603, L1 cache 64KB, L2 cache 256KB, L3 cache 10MB |
| Memory | 2GBx4, Hyundai, 1067MHz, DDR3 |
| Disk | 250GB, 7200 RPM, WD2500AAKX with 16MB cache |
| Flash | 200GB, Intel SSD DC S3700 |

在每个实验之前，作者用虚拟内容重新构建文件系统。对于目录和嵌入引用的方案，复合文件成员会不断更新。对于基于频率的整合，分析是批量执行的，但复合文件每天都在更新。

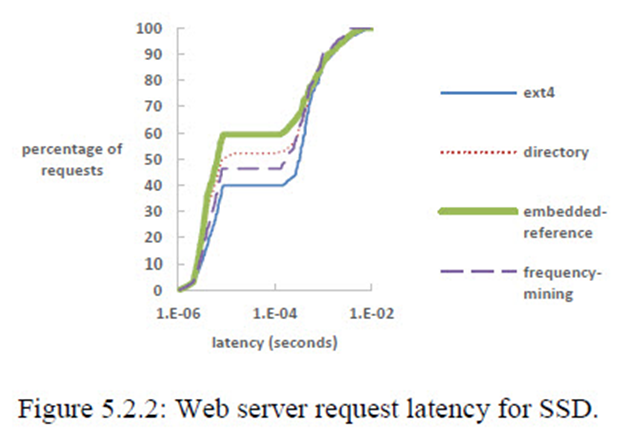
## 5.2 web服务器跟踪检测

HDD性能:图5.2.1显示了一个磁盘的web服务器请求延迟的CDF，从请求发送到请求完成时的时间测量。



本篇论文的根本目的是为了减少经常被一起访问的小文件的IOs数量。在这种情况下，合并元数据使得元数据数量减少，这样的话就可以对所有的子文件元数据进行预取，同时因为复合文件的语义允许跨文件预取，这就导致缓存命中率显著提高。

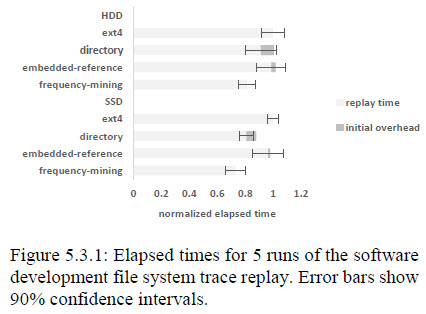
实验结果表明，基于嵌入式引用的整合表现的最好，有62%的请求服务来自缓存，比ext4高20%，因此基于嵌入式引用创建的复合文件可以更准确地捕获访问模式，整个重播时间也减少了20%。基于目录的复合文件还可以提高15%的缓存命中率，这反映了目录本身捕获空间位置的有效性。以频率为基础的整合比基于目录的整合表现得更糟糕。我们检查了日志文件，发现有48%的引用是由爬虫完成的，而剩下的则由其他用户完成的。因此，挖掘算法的分叉交通模式形成了较低的文件分组，减少了收益。



SSD性能:图5.2.2显示SSD的web服务器请求延迟的CDF。相对于磁盘，相对的趋势是相似的，由于SSD的速度，缓存遗漏的请求延迟时间减少了两个数量级。由于主要的性能收益是由较高的缓存命中率和IO避免引起的，所以这20%的好处与底层存储介质无关。

## 5.3 软件开发文件系统跟踪回放

在对软件开发工作负载的跟踪过程中，捕获单个文件系统调用请求的延迟很困难，因为很多都是异步的(例如，写)，并且还有像mmap这样的调用不知道发送到底层存储的请求数。因此，作者用总的经过时间来对结果作总结，其中包括复合文件操作的所有开销，不包括目录和嵌入引用的方案的初始设置成本(图5.3.1)。



**HDD性能:**基于嵌入的基于引用的方案的覆盖率很低，因为许多引用与编译无关。因此，经过的时间更接近ext4。基于目录的整合实现了17%的时间减少，但是基于频率的计划可以达到27%，因为复合文件包括跨目录的文件。

**SSD性能:**不同整合方式的相对性能趋势与HDD相似。类似于web跟踪，增益高达20%。

当比较HDD和SSD实验之间的性能改进差距时，SSD无法实现HDD的11%的性能提升，因为SSD不会带来磁盘寻找开销。

## 5.4 代价

基于目录和嵌入引用的方案:基于目录和嵌入引用的方案需要初始部署成本，以创建基于目录和嵌入式文件引用的复合文件。嵌入引用方案的初始成本取决于文件引用的文件类型的数量。对于工作负载而言，这个成本在1到14分钟之间(图5.3.1)。

至于更新复合文件成员的增量成本，添加成员包括附加到复合文件。删除成员主要涉及元数据更新。复合文件在未使用的分配空间的一半之前不会被压缩。由于跟踪重播的代价已经包括了这个开销，所以这个成本被收益抵消了。

基于频率计算的方案:记录集中开销低于0.6%，记录分析的内存开销在200MB范围内，平均每天有1500万行日志。这种基于频率的计划涉及到从最近的文件引用中学习，并且花了一些时间来收获这个计划的全部好处。

## 5.5 讨论和未来的工作

复合文件可以使用不同的存储介质同时受益于读写负载和读写工作负载，这表明性能的提高主要是由于IOs(~ 20%)的减少。SSD和HDD之间的性能改善差距表明，由于磁盘的减少和修改后的数据布局，性能收益将上升到10%。

总的来说，作者对形成复合文件的方式、合并元数据的性能影响以及复合文件的预抓取产生了兴趣。未来的工作将探索形成复合文件的其他方法，并将它们的相互作用与性能贡献的不同组件进行量化。此外，未来的工作将更全面地探索元数据压缩、并发性和安全性的影响。

# 6. 相关工作

小文件优化:本篇论文虽然研究关注的是逻辑文件和物理元数据的多对一映射，但这项工作与通过减少存储访问数来优化小文件访问的方法密切相关。这个领域的早期工作涉及到将一个文件的i - node和它的第一个数据块(MUL84)和嵌入的i - node在目录中(GAN97)进行搭配。稍后，hFS[ZHA07]使用单独的存储区域来优化小文件和元数据访问。Btrfs[ROD13]将元数据和小文件打包成写在拷贝上的b -树。TableFS[REN13]将元数据和小文件打包到一个表中，并将表项修改的2MB日志作为一个日志结构的合并树组织起来。CFFS通过合并经常访问的文件的i节点来补充许多现有的方法。

在并行和分布式计算领域中也探索了访问子文件区域和巩固元数据的想法，其中多个计算机的cpu需要访问相同的大数据文件[YU07]。Facebook的照片存储[BEA10]利用了这样的观察，即图像的权限基本上是相同的，可以被整合。然而，这些机制是为非常均匀的数据类型而定制的。通过使用不同的方式来形成复合文件，CFFS可以使用具有更丰富的内容和访问语义的子文件。

预抓取:虽然可以发现大量的工作可以提高预抓取，但c - miner[LI04]最接近我们的工作。特别是，c - miner应用了块级的频率序列挖掘来优化文件和元数据块的布局，并改进预取。然而，CFFS减少了经常访问的元数据块的数量，并避免了需要一个大的表来映射物理块的逻辑。此外，该文件系统级的挖掘处理的对象和相关的开销大大减少。DiskSeen[DIN07]结合了磁盘布局的知识来改进预抓取，并且预取可以跨文件和元数据边界。CFFS积极地减少物理元数据项的数量，并改变存储布局以促进顺序预取。Soundararajan等[2008]观察到，通过将高级执行上下文(例如线程、应用程序ID)传递到块层，结果数据挖掘可以生成在并发工作负载下运行更长的预抓取规则。由于CFFS在文件系统级别执行数据挖掘，我们可以使用PIDs和IP地址来分离并发文件引用。然而，CFFS的重点是将逻辑文件映射到物理表示，并且可以采用各种挖掘算法来合并元数据和改进存储布局。

# 7. 结论

作者在本文中介绍了composite - file文件系统的设计、实现和评估，它探索了逻辑文件和元数据的多对一映射。可以用不同的方式配置CFFS来识别经常一起访问的文件，并可以合并它们的元数据。结果显示，在两个实际工作负载下，性能提高了27%。CFFS经验表明，文件和元数据之间的一对一映射的方法是有希望的，并且可以带来许多新的优化机会。