|  |  |
| --- | --- |
| 学号 | 2017202110125 |

**海量存储读书报告**

|  |
| --- |
| **海量存储中文件系统的优化** |

|  |  |
| --- | --- |
| 院（系）名 称 ： | 计算机学院 |
| 专 业 名 称 ： | 网络空间安全 |
| 学 生 姓 名 ： | 涂 子 璇 |

二〇一七年十二月

摘 要

随着互联网的发展，存储的文件数量也呈几何级的增长。传统文件系统已不能满足存储性能的需求。对于文件存储，尤其是海量文件存储的优化已变得越来越重要。本读书报告研究了基于 NAND 的存储设备设计的新型开源 flash 文件系统F2FS和存储小文件的复合文件系统CFFS。

F2FS是一个Linux文件系统，设计用于现代闪存存储设备。文件系统建立在追加记录上，其关键设计决定是考虑到闪存的特性。本文介绍了FFS的主要设计思想，数据结构，算法和性能。在最先进的移动系统上，在合成工作负载下，性能优于EXT4 3.1 x（iozone）和2 x（SQLite）。它可以减少多达40％的实际工作量。在服务器系统上，F2FS的性能比EXT4高。

传统的文件系统优化通常是运用逻辑文件与它们的物理数据元表示一对一映射的方法。这种映射方法导致它失去了某一类优化的可能性，在这类优化方法中这样的映射对被移除。CFFS文件系统是一种支持文件到数据元的多对一的映射方式的复合文件的文件管理系统，并且探索了不同映射策略的设计空间。在不超过网络服务器和软件开发工作负载的情况下，相比传统文件系统这种文件系统有27%的性能提升。

关键词：闪存友好文件系统；复合文件系统；文件系统优化；

目 录

[1 绪论 1](#_Toc500666721)

[1.1 基于NAND的flash文件系统研究背景 1](#_Toc500666722)

[1.2 复合文件系统研究背景 2](#_Toc500666723)

[2 相关知识补充 3](#_Toc500666724)

[2.1 日志结构化文件系统（LFS） 3](#_Toc500666725)

[2.2 闪存文件系统 3](#_Toc500666726)

[2.3 FTL优化 4](#_Toc500666727)

[2.4 小文件优化 4](#_Toc500666728)

[2.5 预读取 4](#_Toc500666729)

[3 文件系统设计与实现 6](#_Toc500666730)

[3.1 F2FS的设计与实现 6](#_Toc500666731)

[3.1.1 磁盘上布局 7](#_Toc500666732)

[3.1.2 文件结构 8](#_Toc500666733)

[3.1.3 目录结构 9](#_Toc500666734)

[3.1.4 多头记录 9](#_Toc500666735)

[3.1.5 数据清洗 11](#_Toc500666736)

[3.1.6 自适应日志记录 12](#_Toc500666737)

[3.1.7 检查点和恢复 12](#_Toc500666738)

[3.2 CFFS的设计与实现 14](#_Toc500666739)

[3.2.1 数据表示 14](#_Toc500666740)

[3.2.2 元数据表示和操作 15](#_Toc500666741)

[3.2.3 小文件的合并方式 17](#_Toc500666742)

[4 总结 19](#_Toc500666743)

[参考文献 20](#_Toc500666744)

# 绪论

本章介绍改进这两个种类的文件系统的背景。F2FS（Flash Friendly File System，闪存友好文件系统）是基于NAND的flash文件系统，它的目的是为了优化具有通用模块接口的闪存器件的性能和使用寿命。大部分文件系统管理优化技术保留逻辑文件与它们的物理数据元表示的一对一映射，CFFS（The Composite-file File System，复合文件的文件系统）是针对这种严格的映射方式使得基于此的文件系统成为了一类性能优化方法的盲区而设计出来的文件。

## 基于NAND的flash文件系统研究背景

NAND闪存在智能手机，平板电脑和MP3播放器等各种移动设备中得到了广泛的应用。此外，服务器系统开始利用闪存设备作为其主存储器。尽管闪存具有广泛的用途，但它有一些局限性，如在写入之前需要擦除，需要在擦除块上依次写入擦除块，并限制每个擦除块的写入周期。

随着存储需求的增长，使用具有通过专用控制器连接的多个闪存芯片的“解决方案”越来越普遍。在控制器上运行的固件（通常称为FTL（闪存转换层））解决了NAND闪存的限制，并提供了通用块设备抽象。这种闪存存储解决方案的例子包括eMMC（嵌入式多媒体卡），UFS（通用闪存）和SSD（固态驱动器）。通常，这些现代闪存存储设备比硬盘驱动器（HDD），其机械对等设备显示出低得多的访问延迟。当涉及随机I/O时，SSD的性能比HDD好几个数量级。

然而，在闪存存储设备的某些使用条件下，NAND闪存媒体的特性表现出来。例如频繁的随机写入SSD会导致底层媒体的内部碎片化[1]，并降低SSD的持续性能。研究表明，随机写入模式是相当普遍，它是在移动设备上更繁重的资源受限的闪存解决方案。除非仔细处理，否则在现代工作负载中频繁的随机写入和刷新操作可能会严重增加闪存设备的I / O延迟并缩短设备使用寿命。

随机写入的有害影响可以通过日志结构文件系统（LFS）方法和写时复制策略来降低[2]。然而没有考虑闪存设备的特性，在性能和设备寿命方面不可避免地不理想。我们认为传统的硬盘驱动器文件系统设计策略（尽管有益）没有充分利用和优化NAND闪存介质的使用。

在本文中，我们介绍了F2FS（Flash Friendly File System）的设计和实现，F2FS是一种针对现代闪存存储设备优化的新文件系统。F2FS是这是第一个公开且广泛使用的文件系统，它的目的是为了优化具有通用模块接口的闪存器件的性能和使用寿命。

## 复合文件系统研究背景

文件管理系统优化是一个被充分研究的领域。然而，大部分优化技术（例如：缓存，更好的数据布局）保留逻辑文件与它们的物理数据元表示的一对一映射（即在UNIX平台下，每个文件与它们自己的i-node联系在一起）。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依赖这种结构，比如VFS API，预读取和数据元缓存。但是，这种严格的映射方式使得基于此的文件系统成为了一类性能优化方法的盲区。

由此一种基于复合文件的文件管理系统被设计出来。在此系统中，多个逻辑文件可以被编组打包然后与一个inode节点联系在一起。由于一起被访问的多个文件分享相似的数据元分区，因此这种方案是可行的，这些数据元分区可以被精简。所以，CFFS能够减少对存储部件的数据元访问次数。访问小文件时，数据元访问是主要访问代价的来源。而且在现代工作负载中仍然是文件引用的大部分情况。

# 相关知识补充

本章介绍与文件系统优化的关键知识。前三个知识点是三种与本文有关的日志结构文件系统，针对闪存的文件系统以及针对FTL的优化。后两个是与复合文件系统有关的知识点。

## 日志结构化文件系统（LFS）

从最初的LFS提案开始，人们已经在日志结构文件系统（HDD）上做了大量的工作[2]。一些学者提出了一种孔洞堵塞方法，其中受害者段的有效块被移动到洞，即其他脏段中的无效块[3]。适应性清除政策，在正常的采伐政策和基于成本效益评估的堵漏政策之间进行选择[4]。线程化日志在高度利用的体积中提供了更好的性能[5]。 F2FS已经根据以前的工作和现实世界的工作负载和设备进行了调整。

一些研究集中于分离冷热数据。例如应该区分缓冲区缓存中的活动数据和非活动数据，而不是将它们写入单个日志并在清理过程中将其分开[6]。这种方法通过监视访问模式来确定哪些数据是活动的。还有一种混合方法：它使用热页面记录来实现高随机写入性能，并覆盖冷页面以降低清除成本[7]。

与使用访问模式的运行时监视的热/冷数据分离方法不同[6]，F2FS使用随时可用的信息（如文件操作（追加或覆盖），文件类型（目录或常规文件） ）和文件扩展名。更复杂的运行时监测方法可以结合到F2FS中来精确跟踪数据温度。

## 闪存文件系统

之前的研究已经为使用原始NAND闪存作为存储器的嵌入式系统提出并实施了许多文件系统。这些文件系统直接访问NAND闪存，同时解决所有的芯片级问题，如磨损均衡和坏块管理。与这些系统不同的是，F2FS的目标是闪存存储设备，配有专用的控制器和固件（FTL）来处理低级任务。这种闪存设备更为普遍。

还有一种文件系统是DFS（直接文件系统）[8]，它利用主机运行FTL的特殊支持，包括原子更新接口和非常大的逻辑地址空间，以简化文件系统设计。然而，DFS仅限于特定的闪存设备和系统配置，并不是开源的。

## FTL优化

目前在FTL层面上提高随机写入性能的工作很多，与F2FS共享一些设计策略。 大多数FTL使用日志结构更新方法来克服闪存的不可覆盖限制。页面映射FTL通过在运行时监视访问，基于更新频率对数据进行聚类[9]。为了减少大页面映射表的开销[10]，一种方式是动态地将页面映射的一部分加载到工作存储器中，并为具有有限RAM的设备提供页面映射的随机写入优点。

混合映射（或日志块映射）是块映射的扩展，以改善随机写入。它具有比页面映射更小的映射表，而对于具有实质访问局部性的工作负载，其性能可以像页面映射一样好。

## 小文件优化

逻辑文件和物理数据元的多对一映射与通过减少存储访问次数来优化小文件访问的方法紧密相关。这个领域的早期工作涉及到利用第一个数据块来排列文件的inode[11]以及在目录中嵌入inode[12]。之后，hFS[13]利用分离的存储区域优化小文件和数据元访问。Btrfs[14]将数据元和小文件打包进写时复制的b树。TableFS[15]将数据元和小文件打包进一个表格，并刷新记录表格条目修改操作的2MB大小的日志，这些日志被组织成日志结构的合并树。通过联合经常被一起访问的文件的inode，CFFS补充了现有的这些方法。

在并行和分布式计算领域，多台计算机上的处理器需要访问同样大小的数据文件。在此基础上，研究者们也探索了访问子文件域和合并数据元的方法[16]。Facebook的照片存储就利用了图像的访问权限很大程度上是相同的而且可以被合并的观察现象[17]。然而，这些机制是为刚好同类的数据类型制订的。采用不同的方法来组织复合文件，CFFS能够处理内容和访问语义差异性更大的子文件。

## 预读取

在大量改进预读取操作的研究中， C-Miner[18]与本文中复合文件系统的预读取的工作是最接近。具体来说，C-Miner运用块层次的频繁序列挖掘来优化文件和数据元块的布局，改进预读取方法。然而，CFFS减少了频繁访问元数据块的次数，避免了对映射逻辑块与物理块的大表格的需要。除此之外，我们的文件系统层次的挖掘只需处理少得多的对象和与之相关的负载。DiskSeen[19]结合了硬盘布局的知识来改进预读取方法，并且使得预读取方法能够突破文件和数据元的界限。CFFS向前读取地减少了物理数据元项目的数量并且更改了存储布局来促进顺序预读取。他们注意到绕过高层次的执行上下文（例如：线程，应用ID）去到块层，作为结果的数据挖掘能够产生预读取规则，且在并发的工作负载下有更长的运行时间。因为CFFS在文件系统的层次执行数据挖掘，所以我们可以使用PID和IP地址来梳理并发的文件引用。但是，CFFS着重于改变逻辑文件到其物理表示的映射，并且可以采用各种挖掘算法来合并元数据并改进存储布局。

# 文件系统设计与实现

本章对两种文件系统的设计与实现做介绍。与大多数文件系统的介绍相似，本文会介绍这两种文件系统的数据表示方式和文件结构。而针对基于flash文件的特点和针对小文件组合的特点，这两种文件系统各有多日志记录和合并小文件方式等不同的章节。

## F2FS的设计与实现

以下列出了F2FS设计的主要考虑事项：

•Flash友好的磁盘布局（第3.1.1节）。F2FS采用三个可配置的单元：段，节和区域。它从多个单独的区域以段为单位分配存储块。它以章节为单位执行“清理”。引入这些单位是为了与底层FTL的运营单位保持一致，以避免不必要的（但昂贵的）数据复制。

•性能高的目录结构（第3.1.2节）。LFS将数据和索引块写入新分配的空闲空间。如果叶数据块更新（并写入某处），则其直接索引块也应更新。一旦直接索引块被写入，它的直接索引块应该被更新。这种递归的日期导致了一连串的写作，造成了“流浪树”的问题。为了解决这个问题，我们提出了一个新的索引表，称为节点地址表。

•多头记录（3.1.4节）。我们设计了一个有效的热/冷数据分离方案，在记录时间（即块分配时间）期间应用。它同时运行多个活动日志段，并根据其预期的更新频率将数据和元数据附加到不同的日志段。由于闪存存储设备利用媒体并行性，多个活动段可以同时运行而无需频繁的管理操作，因此多个记录相对于单段记录性能下降不显着。

•自适应日志记录（第3.1.6节）。F2FS基本上只在追加记录的基础上随机写入数据。然而，在高存储利用率的情况下，它将日志记录策略改为线程化记录以避免长时间的写入延迟。本质上，线程化日志记录将新数据写入脏段中的空闲空间，而无需在前台进行清理。这种策略适用于现代闪存设备，但在HDD上可能不这样做。

•fsync加速前滚恢复（第3.1.7节）。通过最小化所需的元数据写入和通过高效的前滚机制恢复同步的数据，F2FS优化了小型同步写入以减少fsync请求的延迟。

简而言之，F2FS建立在LFS的概念之上，但是与新的设计考虑因素相比，F2FS和LFS相比有很大的不同。

### 磁盘上布局

F2FS的磁盘数据结构经过精心布置，可以与NANO闪存的组合或管理方式相匹配。如图3.1所示，F2FS将整个卷分成固定大小的段。该段是F2FS中的基本管理单元，用于确定初始文件系统元数据布局。

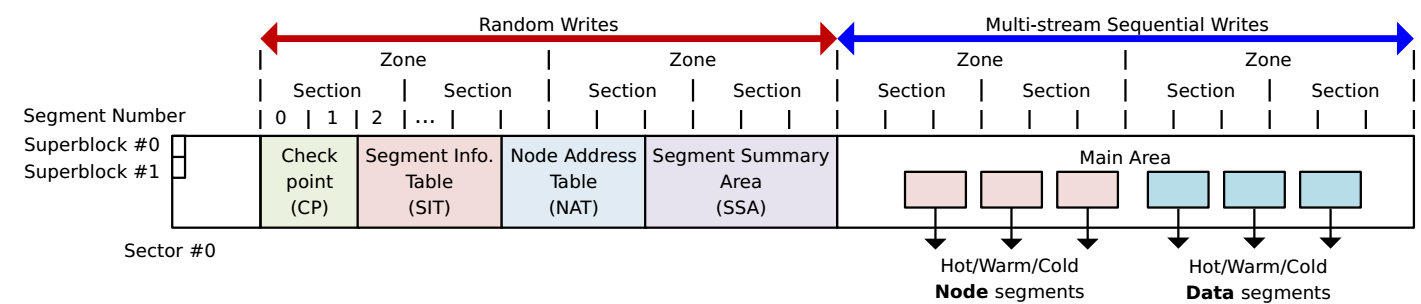


图3.1：F2FS的磁盘布局

一个节由连续的部分组成，而区域由一系列节组成。这些单元在记录和清除过程中非常重要，在第3.2.4节和第3.2.5节中对此进行了进一步讨论。

F2FS将整个卷分成六个区域：

•超级块（SB）具有基本的分区信息和F2FS的默认参数，它们在格式化时间给出并且不可更改。

•检查点（CP）保留文件系统状态，有效的NAT /SIT集的位图（见下文），当前活动段的孤立索引节点列表和摘要条目。一个成功的“检查点包”应该在给定的时间点存储一个一致的F2FS状态，即一个在突然断电事件之后的恢复点（3.2.7节）。CP区域在＃0和＃1这两个段上存储两个检查点包：一个用于上一个稳定版本，另一个用于中间（过时）版本。

•段信息表（SIT）包含每个段信息，例如有效块的数量和“主”区域（见下文）中所有块有效性的位图。检索SIT信息以选择victim区段并在清除过程中识别其中的有效区块。（第3.2.5节）。

•节点地址表（NAT）是定位存储在主区域中的所有“节点块”的块地址表。

•段摘要区域（SSA）存储代表主区域中所有块的所有者信息的汇总条目，例如父索引节点号和其父节点号节点/数据偏移。在清理期间，迁移有效的块之前，SSA条目标识父节点块。

•主区域由4KB大小的块填充。每个块都被分配并定义类型为节点或数据。节点块包含inode或数据块索引，而数据块包含目录或用户文件数据。请注意，一个节不会同时存储数据和节点块。

考虑到上面的磁盘数据结构，让我们来看看如何完成文件查找操作。假设文件“/ dir / file”，F2FS执行以下步骤：（1）通过读取从NAT（节点地址表）获得的块来获得根索引节点;（2）在根索引节点块中，它从它的数据块中搜索一个名为dir的目录项并获得它的inode号;（3）通过NAT将检索到的inode号码转换为物理位置;（4）通过读取相应的块得到名为dir的inode;（5）在目录索引节点中，它标识名为file的目录条目，最后通过对文件重复步骤（3）和（4）获得文件索引节点。实际的数据可以从主区域检索，索引通过相应的文件结构获得。

### 文件结构

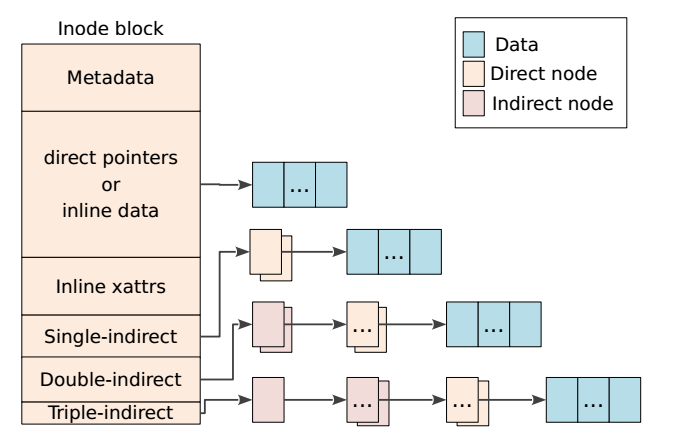
原始的LFS引入的inode映射将inode编号转换为磁盘上的位置。相比之下，F2FS利用扩展索引节点映射的“节点”结构来定位更多的索引块。每个节点块都有一个唯一的标识号“节点ID”。通过使用节点ID作为索引，NAT服务于所有节点块的物理位置。节点块代表三种类型之一：inode，直接节点和间接节点。inode块包含文件的元数据，如文件名，inode号，文件大小，atime和dtime。直接节点块包含数据的块地址，间接节点块具有定位另一节点块的节点ID。

图3.2：F2FS的文件结构

如图3.2所示，F2FS使用基于指针的文件索引与直接和间接节点块来消除更新传播（即“漫游树”问题）。在传统的LFS设计中，如果叶子数据被更新，则其直接和间接指针块被更新图3.2：F2FS的文件结构递归。然而，F2FS只更新一个直接节点块和它的NAT入口，有效地解决了广播树问题。例如，当一个4KB的数据被附加到一个8MB到4GB的文件时，LFS递归地更新两个指针块，而F2FS只更新一个直接节点块（不考虑缓存效应）。对于大于4GB的文件，LFS会再更新一个指针块（总共三个），而F2FS仍然只更新一个。

inode块包含指向文件数据块的直接指针，两个单独间接指针，两个双重间接指针和一个三重间接指针。F2FS支持嵌入式数据和嵌入式扩展属性，嵌入式扩展属性嵌入inode块自身的小型数据或扩展属性。内联减少了空间需求并提高了I/O的表现。请注意，许多系统都有小文件，并有少量的文件被扩展。默认情况下，如果文件大小小于3,692字节，F2FS将激活数据的内联。F2FS在inode块中保留200个字节用于存储扩展属性。

### 目录结构

在F2FS中，4KB目录条目块由位图和两个成对的槽和名称组成。位图告诉每个插槽是否有效。一个插槽携带一个哈希值，inode数量，文件名和文件类型的长度（例如，普通文件，目录和符号链接）。目录文件构造多级哈希表来高效地管理大量的目录项。

当F2FS在目录中查找给定的文件名时，它首先计算文件名的哈希值。然后，它从0层到记录在inode中的最大分配级别递增地遍历所构建的散列表。在每个级别上，它扫描一个内含两个或四个目录项块的容器，造成O（log（目录项级别））的时间复杂度。要更快地找到一个目录条目，方法是按顺序比较位图，散列值和文件名。

当大型目录是首选的（例如，在服务器环境中）时，用户可以配置F2FS来初始化分配多个目录条目的空间。在较低级别的较大哈希表中，F2FS更快地达到目标分区。

### 多头记录

与一个具有很多日志区域的LFS不同，F2FS主要保留六个主要的日志区域，以达到最大化冷热数据分离的效果。F2FS静态地为节点和数据块定义了三个温度级别“热”，“暖”和“冷”’，如表3.1所示。

由于直接节点块更频繁地更新，因此直接节点块被认为比热门节点块更热。间接节点块包含节点ID，只有在添加或删除专用节点块时才写入。目录的直接节点块和数据块被认为是热点，因为与常规文件的块相比，它们具有明显不同的写入模式。满足以下三个条件之一的数据块被认为是冷的：

表3.1:多个活动段中的对象分类

|  |  |  |
| --- | --- | --- |
| **类型** | **温度级别** | **对象** |
| 节点 | 热 | 目录的直接节点块 |
| 暖 | 常规文件的直接节点块 |
| 冷 | 间接节点块 |
| 数据 | 热 | 目录项块 |
| 暖 | 用户的数据块 |
| 冷 | 因数据清除移动的数据块；  用户标记为“冷”的数据块；  多媒体文件数据 |

由于直接节点块更频繁地更新，因此直接节点块被认为比热门节点块更热。间接节点块包含节点ID，只有在添加或删除专用节点块时才写入。目录的直接节点块和数据块被认为是热点，因为与常规文件的块相比，它们具有明显不同的写入模式。满足以下三个条件之一的数据块被认为是冷的：

•因数据清除移动的数据块（参见第3.2.5节）。由于它们在很长一段时间内仍然有效，我们预计它们在不久的将来仍将如此。

•用户标记为“冷”的数据块。这是F2FS支持的扩展的属性操作。

•多媒体文件数据。他们可能会显示一次写入和只读模式。F2FS通过将文件的扩展名与注册的文件扩展名进行匹配来标识它们。

默认情况下，F2FS激活六个日志打开写入。用户可以在安装时将写入流的数量调整为两个或四个，如果这样做被认为在给定的存储设备和平台上产生更好的结果。如果使用六个日志，则每个日志记录段直接对应于表3.1中列出的温度级别。在四个日志的情况下，F2FS结合每个节点和数据类型中的冷日志和温日志。只有两个日志，F2FS分配一个节点，另一个分配数据类型。

F2FS引入了可配置区域来与FTL兼容，以减轻垃圾收集（GC）的开销. FTL算法主要根据关联性分为三个组（块关联，集合关联和完全关联）之间的数据和“日志闪存块”。一旦数据闪存块被分配来存储初始数据，日志闪存块尽可能吸收数据更新，如EXT4中的日记。 对于所有的数据闪存块（完全关联的），或者对于一组连续的数据闪存块（set-associative），对数闪存块（block-associative））。现代FTL采用完全关联或集合关联的方法，以便能够正确处理随机写入。请注意，F2FS使用多头记录并行写入节点和数据块，并且关联的FTL会将分离的块（在文件系统级别中）混合到同一个闪存块中。为了避免这种错位，F2FS将活动日志映射到不同的区域，以在FTL中分离它们。预期这种策略对于集合关联的FTL是有效的。多头记录也是最近提出的“多流”接口的自然匹配。

### 数据清洗

清理是一个过程，回收分散和无效的块，并确保自由段进一步的日志。因为一旦潜在的储存容量已经填满，清除就会不断发生，限制与清除相关的成本对于F2FS（以及一般的任何LFS）的可靠性是非常重要的。在F2FS中，以一个节为单位进行清除。

F2FS以两种截然不同的方式进行清除，即前景和背景。前台清理只有在没有足够空闲部分的情况下才会触发，而内核线程定期唤醒后台进行清理。清除过程需要三个步骤：

1)受害者的选择。清除过程首先开始识别非空白部分中的受害者部分。在LFS清洗过程中有两个著名的受害者选择策略-贪婪策略和成本效益策略。贪婪策略选择一个有效块数量最少的段。直观地说，该策略控制迁移有效块的开销。F2FS采用贪婪策略进行前台清理，以尽量减少应用程序可见的延迟。此外，F2FS保留了一个小的未使用的容量（5％的存储空间），以便清除过程在高存储利用率的情况下有足够的空间。

另一方面，成本效益政策是在F2FS的后台清理过程中进行的。这个政策不仅根据其使用情况而且还选择受害者部分的“年龄”。F2FS通过平均段中段的年龄来推断段的年龄，这又可以从SIT中记录的最后修改时间获得。通过成本收益政策，F2FS可以分离冷热数据。

(2)有效的块识别和迁移。在选择受害者部分之后，F2FS必须快速识别该部分中的有效块。为此，F2FS在SIT中为每个段维护一个有效位图。一旦通过扫描位图来识别所有有效的块，F2FS就从SSA信息中检索包含其索引的父节点块。如果块有效，则F2FS将其迁移到其他空闲日志上。

对于后台清理，F2FS不会发出实际的I / O来迁移有效的块。相反，F2FS将块加载到页面缓存中并将其标记为脏。然后，F2FS将其留在内核工作线程的页面缓存中，以便稍后将其刷新到存储中。这种迁移方式不仅缓解了对前台I/O活动的性能影响，而且还允许将小的块写入进行组合。当正常的I/O或前景清除正在进行时，背景清除不会同时进行。

(3)后期清除过程。在所有有效的块都完成后，受害节被记录为候选节，成为新的空闲部分（在F2FS中称为“空闲之前”部分）。检查点完成后，该部分最终成为空闲部分，将被重新分配。我们这样做的原因是，如果在检查点之前重新使用了预先释放的部分，则文件系统可能会在意外断电发生时丢失之前检查点引用的数据。

### 自适应日志记录

原来的LFS引入了两个日志记录策略，也就是日志记录和线程日志。在正常的日志中，块被写入干净的段，产生严格按顺序写入。即使用户提交了许多随机写入请求，只要存在足够的空闲日志空间，该过程就会将其转换为顺序写入。然而，随着空闲空间缩小到零，这种策略开始遭受很高的清除开销，导致性能下降严重。另一方面，线程化日志记录将块写入现有脏段中的“洞”（无效，废弃空间）。此策略不需要清理操作，但是会触发随机写入，因此可能会降低性能。

F2FS根据文件系统状态动态实现策略并进行切换。具体来说，如果存在多于k个清除部分，其中k是预定义阈值，则启动正常记录。否则，线程日志被激活。k默认设置为5％，可以进行配置。

当存在散布的漏洞时，线程化记录有可能导致不希望的随机写入。尽管如此，这种随机写入通常比现场更新文件系统显示更好的空间局部性，因为当F2FS在其他脏段中搜索更多之前，脏段中的所有空洞都会先填充。

### 检查点和恢复

F2FS实现检查点功能，以便在突然断电或系统崩溃时提供一致的恢复点。无论何时需要在同步，镜像和前台清理等事件中保持一致的状态，F2FS将按如下方式触发检查点过程：（1）刷新页面缓存中的所有脏节点和目录条目块;(2)中止普通文字活动，包括系统调用，如create,unlink和mkdir指令; (3)将文件系统元数据，NAT,SIT和SSA写入磁盘上的专用区域; (4)F2FS将包含以下信息的检查点包写入CP区域：

•页眉和页脚分别写在包的开头和结尾。F2FS在页眉和页脚中保留一个在创建检查点时递增的版本号。版本号在安装时间内区分两个录制包之间的最新稳定包;

•NAT和SIT位图指示包括当前包的一组NAT和SIT块;

•NAT和SIT期刊包含少量的重新编辑，彻底修改NAT和SIT的条目以避免频繁的NAT和SIT更新;

•活动段的摘要块包含在将来刷新到SSA区域的内存SSA块中;

•孤立块保留“孤立inode” 信息。如果inode在关闭之前被删除（例如，当两个进程打开一个公共文件，一个进程删除它时会发生这种情况），它应该被注册为一个孤立的inode，以便F2FS在突然关闭电源后可以恢复。

**回滚恢复**：突然断电后，F2FS回滚到最新的一致检查点。为了在创建新包的同时至少保留一个稳定的检查点包，F2FS维护两个检查点包。如果检查点包在页眉和页脚中具有相同的内容，则F2FS认为它有效。否则，该包被丢弃。

同样，F2FS也管理两组NAT和SIT块，由每个检查点包中的NAT和SIT位图区分。在检查点写入更新的NAT或SIT块时，F2FS将它们写入两个集合中的一个集合，然后将该位图标记为指向它的新集合。

如果少量NAT或SIT条目频繁更新，则F2FS将编写多个4KB大小的NAT或SIT块。为了减轻这种开销，F2FS在检查点包中实现了NAT和SIT日志。这种技术减少了I / O数量，因此也减少了检查点的延迟。在安装时的恢复过程中，F2FS通过检查页眉和页脚来搜索有效的检查点包。如果两个检查点包都是有效的，那么F2FS通过比较它们的版本号来选择最新的一个。一旦选择了最新的有效检查点包，它将检查是否存在孤立inode块。如果是这样的话，它将阻止它们引用的所有数据块，最后也释放孤立的inode。然后，在前滚恢复过程成功完成之后，F2FS启动文件系统服务，并使用其位图所引用的一组一致的NAT和SIT块，如下所述。

**前滚恢复：**像数据库（如SQLite）这样的应用程序经常将小数据写入文件，并进行fsync同步以保证持久性。支持fsync同步的一个简单的方法是触发检查点和回滚模型恢复数据。但是，这种方法导致性能较差，因为检查点包括写入与数据库无关的所有节点和分区块。

F2FS实现了一个高效的前滚恢复机制，以提高fsync性能。关键的想法是仅写入数据块及其直接节点块，不包括其他节点或F2FS元数据块。为了在回滚到稳定的检查点之后有选择地查找数据块，F2FS在直接节点块内保留一个特殊的标志。

F2FS执行前滚恢复如下。如果我们将最后一个稳定检查点的日志位置表示为N，（1）F2FS收集具有位于N + n中的特殊标志的直接节点块，同时构造它们的节点信息的列表。n是指自上一个检查点以来更新的块数。（2）通过使用列表中的节点信息，它将最近写入的名为N-n的节点块加载到页面缓存中。（3）然后比较N-n和N + n之间的数据索引。（4）如果检测到不同的数据索引，则刷新存储在N + n中的新索引的高速缓存节点块，最后将其标记为脏。一旦完成前滚恢复，F2FS将执行检查点操作，将整个内存中的更改存储到磁盘。

## CFFS的设计与实现

CFFS引入了一个称为复合文件（composite file）的内部物理表示，该文件保存经常一起访问的小文件的内容。复合文件对最终用户是不可见的，并且与在小文件中共享的单个复合inode相关联。存储在小文件inode中的原始信息删除了重复数据，而且存储为复合文件的扩展属性。各个小文件的元数据属性仍然可以被重建，检查和更新，使得传统访问语义（例如，类型，权限，时间戳）不变。扩展属性还记录了单个小文件在复合文件内的位置。通过这种呈现方式，CFFS可以将物理复合文件转换为逻辑文件。

将哪些文件组合成一个复合文件是一个主要依赖于工作负载的策略决策。比如说，CFFS已经被配置了三种方式。第一种方案是基于目录的合并，目录中的所有文件（不包括子目录）形成一个复合文件。第二种方案是嵌入式参考整合，该方法是提取文件内容中的文件引用以识别可以形成组合文件。三是基于频次挖掘的整合，通过设置频次挖掘对文件参考进行分析，这样将被经常一起访问的文件形成复合文件。

一个复合文件利用了传统的VFS预取机制，因为整个复合文件可以以类似于通过将小数据块组合成更少的较大块而获得FFS文件系统的优点的方式被预取为一个单元。

### 数据表示

复合文件的内容是通过串联小文件（称为子文件）形成的。 复合文件中的所有子文件共享相同的inode（信息节点），以及间接块，双重间接块等。复合文件的最大尺寸限制不是一个问题，因为复合文件被设计为对小文件进行分组。如果子文件大小的总和超过了最大文件大小限制，我们可以使用多个复合文件。

通常，复合文件中的第一个子文件是入口点，其入口将触发预取其余子文件。 例如，当浏览器访问一个html文件时，它会加载一个css文件和flash脚本。该html文件可以作为这个三子文件合成文件的入口点和预取触发器。对于基于频率的合并，子文件的排序反映了它们如何被访问。尽管同一组文件可能具有不同入口点的不同访问模式，但是数据布局基于最普遍的访问模式。

### 元数据表示和操作

复合文件创建：创建复合文件时，CFFS分配一个inode并复制连接子文件的内容作为其数据。该复合文件会将各个子文件的复合文件偏移量和大小以及重复的inode被删除的信息记录到其扩展属性中。 然后原始子文件被截断，其目录条目被重新映射到复合文件的inode，inode还包括子文件ID和其原始inode的重新分配。因此，最终用户仍然可以感知命名空间中的单个逻辑文件，而单个子文件仍然可以被定位（图3.3）。

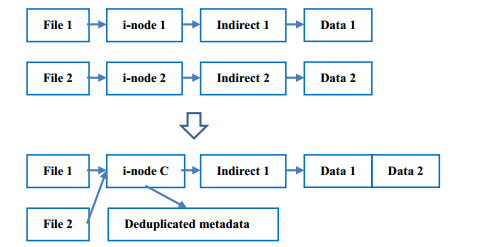


图3.3：两个原始文件（顶部）创建成内部组合文件（底部）

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。

权限：在文件打开时，首先根据复合i-node检查权限测试。 如果失败，则不需要进一步检查。否则，如果子文件具有作为扩展属性而被存储的不同权限，则将再次检查权限。因此，复合i-node将具有包含所有子文件权限的最广泛的权限。例如，如果在复合文件中，我们有一个只读的子文件A和一个可写的子文件B，复合i-node的权限将是读/写。但是，打开子文件A并写入时，扩展属性中的只读权限限制将把写入权限作为违规处理。

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。

大小：对于数据访问，通过扩展属性中编码的子文件偏移量和大小对偏移量进行转换和绑定检查。复合文件的大小是复合文件的长度，可以大于其子文件的总大小。例如，如果复合文件中间的子文件被删除，则该区域将被释放，而不会更改复合文件的大小。

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i-node号码称为CFFS唯一ID（CUID）。

子文件查找和重命名：如果目录中的名称映射到CUID，则可以通过子文件ID查找子文件的属性。重命名将继续进行，就如同这个CUID部署CFFS系统中的一个i-node号码一样。由于将子文件移入和移出复合文件将改变其CUID，因此我们需要存储后台指针，以更新映射到CUID的所有名称。

CUID中的更改可能会破坏由其i-node唯一标识文件的应用程序（例如，备份）。 然而，当今的文件系统也可能导致不同的文件在不同的时间共享相同的i-node号码; CFFS设计体现了应用程序不应该假设i节点号码是文件的唯一属性的这一点。

子文件和子文件成员更新：将子文件添加到组合文件时，它会附加到组合文件。从复合文件中删除子文件时，复合文件中相应的数据区域在扩展属性中被标记为已释放。

子文件打开/关闭操作：对子文件的打开/关闭调用与对复合文件的打开/关闭调用相同，伴随着文件位置指针的转换。

子文件写入操作：和传统文件系统的处理方式相同，CFFS系统的子文件在原有位置更新。但是，如果更新涉及在复合文件的中间增长子文件，并且在子文件末尾没有可用空间，则移动更新子文件到复合文件的末尾。该方案利用潜在的时间局部性，即不断增长的子文件可能在不久的将来再次增长。

硬链接：目录中的不同名称可以映射到相同的i-node编号或CUID。

空间压缩：当复合文件的一半区域内不包含有用数据时，复合文件会压缩自己的空间。

并发更新到复合文件中的子文件：对复合文件中的子文件的并行更新（几个子文件一起更新）与对普通文件的并发更新具有相同的语义。为避免死锁，检测到涉及并发更新的文件可能必须提取到多个常规文件中（我的理解是把几个并发更新的子文件剔除到复合文件之外变成几个普通的文件）。

锁定和一致性：CFFS不支持群集，但有可能实现子文件锁定子系统。

### 小文件的合并方式

鉴于传统文件系统围绕目录的根深蒂固的空间局部优化，目录是文件访问模式和用于形成组合文件的良好近似。 目前，这个合并计划不包括子目录。

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。

基于嵌入式参考的整合：

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。 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。 类似的想法适用于编译。 我们可以从Makefile中提取依赖关系规则，并合并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可以定期通过修改的文件进行筛选以协调复合文件成员。

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。 另外，它需要知道特定的文件格式。

基于频率挖掘的合并：

在对基于频率挖掘的整合的研究中，我们使用了Apriori算法的一个变种。关键的是，如果经常访问一组文件，它的子集也必须是（Apriori属性）。 图3.4说明了对文件A，B，C，D和E的访问流的算法。

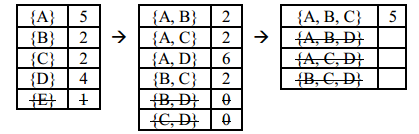


图3.4：Apriori算法为文件参考流E，D，A，D，A，D，A，B，C，A，B，C，A，D标识经常访问的文件集的步骤。

第一遍：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。

第二遍：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。 只要在访问文件B之后访问文件A，我们就会增加文件集{A，B}的计数，反之亦然。 计数小于阈值的集被删除（例如，{B，D}）。

第三遍：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}等文件集被删去，因为在第二遍中{B，D}被消除了。

终止：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，我们返回集{A，B，C}和{A，D}作为两个经常访问的集合。 诸如{A，B}的集合被删除，因为它们是{A，B，C}的子集。

变种：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）。

# 总结

F2FS是为现代闪存存储设备设计的全面的Linux文件系统，预计将在业界广泛采用。 本文描述了F2FS的关键设计和实现细节。 我们的评估结果强调了我们的设计决策和折衷方案如何带来性能优势，优于其他现有的文件系统。 F2FS相当年轻，它在2012年底被整合到Linux内核3.8中。我们预计新的优化和功能将不断添加到文件系统中。

CFFS是一种基于复合文件的文件系统的设计、实现以及评估。此文件系统探索了逻辑文件到其数据元的多对一映射。CFFS可以以不同方式配置，来识别经常被一起访问的文件，并且合并它们的数据元。CFFS的经验表明拆分文件到数据元的一对一映射的方法是有前景的并且可以带来很多新的优化机会。

参考文献

[1] C. Min, K. Kim, H. Cho, S.-W. Lee, and Y. I. Eom. SFS: Random write considered harmful in solid state drives. In Proceedings of the USENIX Conference on File and Storage Technologies (FAST), pages 139–154, 2012.

[2] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. ACM Transactions on Computer Systems (TOCS),10(1):26–52, 1992.

[3] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID hierarchical storage system. ACM Transactions on Computer Systems (TOCS), 14(1):108–136, 1996.

[4] J. N. Matthews, D. Roselli, A. M. Costello, R. Y. Wang, and T. E. Anderson. Improving the performance of log-structured file systems with adaptive methods. In Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)

[5] World Health Organization. Factors regulating the immune response: report of WHO Scientific Group[R]. Geneva: WHO, 1970.

[6] Y. Oh, E. Kim, J. Choi, D. Lee, and S. H. Noh. Optimizations of LFS with slack space recycling and lazy indirect block update. In *Proceedings of the Annual Haifa Experimental Systems Conference*, page 2, 2010.

[7] W. Wang, Y. Zhao, and R. Bunt. Hylog: A high performance approach to managing disk layout. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 144–158, 2004.

[8] W. K. Josephson, L. A. Bongo, K. Li, and D. Flynn. DFS: A file system for virtualized flash storage. ACM Transactions on Storage (TOS), 6(3):14:1– 14:25, 2010.

[9] A. Gupta, Y. Kim, and B. Urgaonkar. DFTL: a flashtranslation layer employing demand-based selective caching of page-level address mappings, volume 44. ACM, 2009.

[10] M.-L. Chiang, P. C. Lee, and R.-C. Chang. Using data clustering to improve cleaning performance for flash memory. *Software-Practice and Experience*, 29(3):267–290, 1999.

[11] Mullender S, Tanenbaum. Immediate Files,*Software Practice and Experience*, 14(4):365-368, 1984.

[12] Ganger GR, Kaashoek MF. Embedded Inode and Explicit Grouping: Exploiting Disk Bandwidth for Small Files. Proceedings of the USENIX 1997Annual Technical Conference (ATC), 1997

[13] Zhihui Zhang, Kanad Ghose. hFS: A Hybrid File System Prototype fr Improving Small File and metadata Performance, Proceedings of the 2nd ACMSIGOPS/EuroSys European Conference onComputer Systems, 2007.

[14]Rodeh O, Bacik J, Mason C. BTRFS: The Linux B-Tree File System. *ACM Transactions on Storage (TOS)*, 9(3), Article No. 9, 2013.

[15]Kai Ren, Garth Gibson. TABLEFS: Enhancing Metadata Efficiency in the Local File System, *Proceedings of the 2013 USENIX Annual Technical Conference (ATC)*, 2013

[16] Yu W, Vetter J, Canon RS, Jian S. Exploiting Lustre File Joining for Effective Collective IO, *Proceedings of the 7th International Symposium on Cluster Computing and the Grid*, 2007.

[17]Beaver D, Kumar S, Li HC, Vajgel P. Finding a Needle in Haystack: Facebook’s Photo Storage. *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[18] 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.

[19] 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