针对闪存优化的文件系统：F2FS

院（系）名 称 ： 计算机学院

专 业 名 称 ： 计算机应用

学 生 姓 名 ： 黎圣

学 生 学 号 ： 2017202110037

二〇一七年十二月

目录

1 介绍 **1**
1.1 论文组织结构 2

2 F2FS 的设计与实现 **2**
2.1 磁盘布局 2
2.2 文件结构 3
2.3 目录结构 4
2.4 多头记录 4
2.5 清理操作 5
2.6 自适应日志记录 6
2.7 检查点和恢复 6
2.7.1 回滚恢复 7
2.7.2 前滚恢复 7

3 实验部分 **8**
3.1 实验设置 8
3.2 实验结果 8
3.2.1 在移动系统上的性能 9
3.2.2 在服务器系统上的性能 10
3.2.3 多头记录影响 11
3.2.4 清理成本 12
3.2.5 自适应日志性能 13

4 相关工作 **14**
4.1 日志结构文件系统 (LFS) 14
4.2 闪存文件系统 15
4.3 FTL 优化 15

5 总结和展望 **15**
参考文献 15

针对闪存优化的文件系统：F2FS

计算机应用 黎圣
2017202110037

摘要

F2FS(Flash Friendly File System) 是专门为基于 NAND 的存储设备设计的新型开源 flash 文件系统。特别针对 NAND 闪存存储介质做了优化设计。F2FS 选择日志结构文件系统方案，并使之更加适应新的存储介质 (NAND)。本文介绍了 F2FS 的主要设计思想，数据结构，算法和性能。

F2FS 在移动设备上，在多种工作负载情况下，性能比 EXT4 高出 3.1 倍 (iozone) 和 2 倍 (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 2.5 倍 (SATA SSD) 和 1.8 倍 (PCIe SSD)。

1 介绍

NAND 闪存已被广泛应用于智能手机，平板电脑和 MP3 播放器等各种移动设备中。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，如先写先擦除的要求，需要在擦除块上依次写入擦除块，以及每个擦除块有限的写入周期等缺点。

早期，许多消费类电子设备直接利用“裸”NAND 闪存搭载平台，随着存储需求的增长，使用多个闪存芯片“解决方案”越来越普遍专用控制器。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示例包括 eMMC(嵌入式多路复用器) timedia 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)，它们的机械对应物显示出更低的访问延迟。就随机 I/O 而言，SSD 比 HDD 要好几个数量级。

然而，在闪存存储设备的一定使用条件下，NAND 闪存介质的特性就体现出来了。例如，Min 等人观察到随机写入固态硬盘会导致底层媒体的内部碎片，并降低可用的固态硬盘性能。研究表明，随机写入模式对移动设备上资源受限的闪存解决方案是相当普遍的，甚至更为重要。Kim 等人量化 Facebook 的移动应用程序问题 150%和 WebBench 注册 70%以上的随机写入比顺序写入。此外，超过 80%的 I/O 是随机的，超过 70%的随机写入由 fsync 通过 Facebook 和 Twitter 等应用程序触发。这些特定的 I/O 模式来自 SQLite 在这些应用程序中的主要用途。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导致闪存设备的 I/O 延迟增加，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/或写时复制策略来降低。例如，人们可能会预期 BTRFS 和 NILFS2 等文件系统在 NAND 闪存 SSD 上表现良好; 不幸的是，他们不考虑闪存存储设备的特性，在性能和设备寿命方面不可避免地是次优的。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，我们介绍了针对现代闪存存储设备优化的新文件系统 F2FS 的设计和实现。据我们所知，F2FS 是第一个公开和广泛使用的文件系统，它是从头开始设计的，以优化具有通用块接口的闪存设备的性能和使用寿命。本文介绍了其设计和实现。

以下列出了 F2FS 设计的主要考虑事项：

Flash 友好的磁盘布局 (第 2.1 节) F2FS 采用三个可配置单位：分段，区域和区域。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。

成本效益指数结构 (第 2.2 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，其内部直接索引块应该被更新。这种递归的更新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，F2FS 提出了一个新的索引表，称为节点地址表。

多头记录 (2.4 节) F2FS 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独立的日志段。由于闪存存储设备利用媒体并行性，因此多个活动段可以同时运行而无需频繁的管理操作，由于多个日志记录 (相对于单段日志记录) 而言性能下降是微不足道的。

自适应日志记录 (第 2.6 节) F2FS 基本上建立在追加记录上，将随机写入转换为顺序写入。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。

带前滚恢复的 fsync 加速 (第 2.7 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，LFS 提案有很大的不同。我们已经将 F2FS 作为 Linux 文件系统实施，并将其与两个最先进的 Linux 文件系统 EXT4 和 BTRFS 进行比较。我们也评估了 NILFS2，这是 LFS 在 Linux 中的一个替代实现。我们的评估考虑了两个一般分类的目标系统：移动系统和服务器系统。在服务器系统的情况下，我们研究 SATA SSD 和 PCIe SSD 上的文件系统。我们获得和在这项工作中呈现的结果突出了 F2FS 的整体理想的性能特征。

1.1 论文组织结构

在本文的其余部分，本文的组织结构如下：第二节首先描述了 F2FS 的设计和实现。第三节提供实验结果和分析讨论。第四节描述本文相关工作。第五部分对全文进行总结，并且对后续的研究方向进行展望。

2 F2FS 的设计与实现

2.1 磁盘布局

F2FS 的磁盘数据结构经过精心布局，以便与底层 NAND 闪存的组织和管理方式相匹配。如图 1所示，F2FS 将整个卷分成固定大小的段。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。

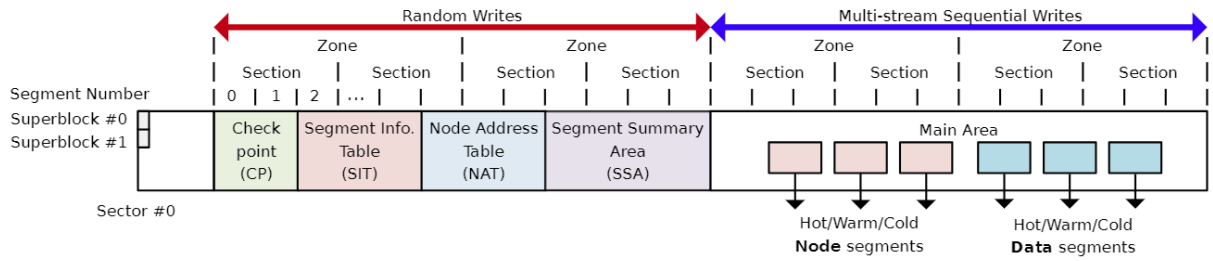


图 1: F2FS 的磁盘布局

节由连续的部分组成，区由一系列节组成。这些区域设置在记录和清洁过程中非常重要，在第 2.4 节和第 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 (一个突然断电事件后的恢复点) 存储一致的 F2FS 状态 (2.7 节)。CP 区域在两个段 (# 0 和# 1) 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) 包含每段信息，如“主”区 (见下文) 中所有块有效性的有效块数和位图。检索 SIT 信息以选择受害人群并在清理过程中识别其中的有效区块 (第 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 (SSA) 存储代表主区中所有块的所有者信息 (例如，父节点号及其节点/数据偏移量) 的摘要试图。在清理期间迁移有效块之前，SSA 条目标识父节点块。

主区域 填充 4KB 区块。每个块都位于同一地点，并被键入为节点或数据。节点块包含 inode 或数据块的索引，而数据块则包含目录或用户文件数据。这里需要注意的是，一个部分不会同时存储数据和节点块。

给定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 获得的地址块来获得根 inode; (2) 在根索引节点块中，从其数据块中搜索名为 dir 的目录条目并获得其索引节点号; (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，标识目录条目 namedfile，最后通过重复文件步骤 (3) 和 (4) 获得文件索引节点。实际数据可以利用相应的文件结构索引从主区域获得。

2.2 文件结构

原始的 LFS 引入的 inode 映射将 inode 号码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，例如文件名，inode 编号，文件大小，atime 和 dtime。直接节点块包含数据的块地址，而间接节点块具有定位另一节点块的节点 ID。

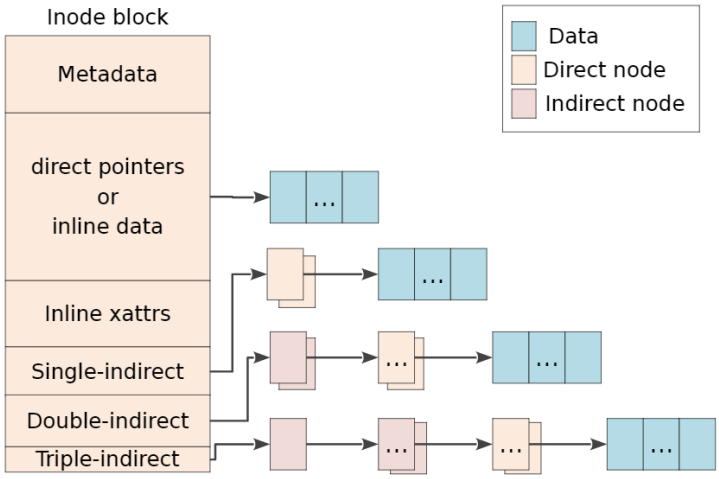


图 2: F2FS 的文件结构

如图 2所示，F2FS 使用基于指针的文件索引与直接和间接节点块消除更新传播 (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递归地更新。然而，F2FS 只更新一个直接节点块和它的 NAT 入口，有效地解决了漫游树问题。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 inline 数据和 inline 扩展属性，这些属性嵌入 inode 块本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 性能。请注意，许多系统都有小文件和少量的扩展属性。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。

2.3 目录结构

在 F2FS 中，一个 4KB 目录项块由一个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。一个时隙携带散列值，索引节点号，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有效地管理大量目录项。

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 级到 inode 中记录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，它扫描一个包含两个或四个目录入口块的桶，导致 O(log(＃号目录项)) 的复杂性。要更快地找到一个目录项，它将按顺序比较位图，散列值和文件名。

当首选大型目录时 (例如，在服务器环境中)，用户可以配置 F2FS，以便为多个登记项初始分配空间。在较低级别的较大哈希表中，F2FS 更快地到达目标目录项。

2.4 多头记录

与 LFS 使用一个大日志区域不同，F2FS 主要保留六个主要的日志区域来做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热度-热，冷和冷-级别，如表 1所示。

直接节点块被认为比间接节点块更热，因为它们被更频繁地更新。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

表 1: 多个活跃段的对象划分

Type	Temp.	Objects
Node	Hot	目录的直接节点
	Warm	常规数据块的直接节点
	Cold	间接节点块
Data	Hot	目录项块
	Warm	用户数据块
	Cold	被清理的数据块; 用户指定的冷数据; 多媒体数据

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：

(1) 由于清理操作数据块被移除 (参见第 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。

(2) 用户标记为“冷”的数据块 。为此，F2FS 支持扩展的属性操作。

(3) 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数据流的数量调整为两个或四个，如果这样做被认为可以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 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。第 3.2.3 节将考察记录头的数量如何影响数据分离的有效性。

F2FS 引入了可配置区域来与 FTL 兼容，以减轻垃圾收集 (GC) 开销。根据数据与“日志闪存块”之间的关联性，FTL 算法主要分为块关联，集合关联和完全关联三类。一旦数据闪存块被分配用于存储初始数据，日志闪存块将尽可能地同步数据更新，如 EXT4 中的日志。对于所有数据闪存块 (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 (块关联))。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。预计这种策略对集合联合 FTL 是有效的。多头记录也是最近提出的“多流”接口。

2.5 清理操作

清理是一个回收分散、无效块，确保未来日志足够的空闲段的过程。因为一旦潜在的存储容量已经被清理，清理就会不断发生，限制与清理相关的成本对于 F2FS(以及任何一般的 LFS) 的持续性能是非常重要的。在 F2FS 中，清洁是以一个部分为单位完成的。

F2FS 以前后台和后台两种不同的方式执行清理。前台清理只有在没有足够空闲部分的情况下才会触发，而内核线程会定期唤醒以在后台进行清理。清洁过程需要三个步骤：

(1) 被清理对象的选择。 清洁过程首先开始识别非空白部分中的需要清理部分。在 LFS 清洗过程中有两个著名的选择策略 - 贪婪和成本效益。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 还保留了一小部分未使用的容量 (5%的存储空间通过故障排除)，以便在高存储利用率的情况下，清理过程有足够的空间进行充分的操作。第 3.2.4 节研究利用水平对清洁成本的影响。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 有机会分离冷热数据。

(2) 有效的块识别和迁移。 在选择受害者部分后，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。如果块是有效的，F2FS 将它们迁移到其他空闲日志。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 将它们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。这种懒惰的迁移不仅缓解了对前台 I / O 活动的性能影响，而且还允许将小写入结合起来。当正常的 I / O 或前台清洁正在进行时，后台清洁不起作用。

(3) 后期清洁操作。 在所有有效的块被清理后，受害者部分被注册为候选者，成为新的空闲部分 (在 F2FS 中称为“提前清理”节)。检查点完成后，该部分最终成为空闲节，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清理的节，则文件系统可能会在发生意外断电时丢失之前检查点引用的数据。

2.6 自适应日志记录

原始的 LFS 引入了两个日志记录策略，即正常日志记录和线程日志记录。在正常的日志记录中，块被写入干净的段，产生严格的顺序写入。即使用户提交了许多随机写入请求，只要有足够的空闲日志空间，这个过程就会将它们转换为顺序写入。然而，随着空闲空间缩小到零，这项政策开始遭受高昂的清理费用，导致性能严重下降 (在苛刻的条件下量化到 90%以上，参见第 3.2.5 节)。另一方面，线程化日志记录将块写入现有脏段中的空洞 (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，如果有多于 k 个清理区段，其中 k 是预定义的阈值，则启动正常日志记录。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。但是，这样的随机写入通常比在原地更新文件系统中显示更好的空间局部性，因为在 F2FS 在其他脏段中搜索更多内容之前，脏段中的所有空洞都会先填充。Lee 等人表明，闪存存储设备显示出更强的空间局部性的随机写入性能。F2FS 优雅地放弃了正常的日志记录，并转向了线程化日志记录，以获得更高的持续性能，如 3.2.5 节所示。

2.7 检查点和恢复

F2FS 具有检查点功能，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，卸载和前台清理等事件中保持一致的状态，F2FS 将触发一个检查点过程，如下所示：

(1) 刷新页面缓存中的所有脏节点和分区块;

(2) 中止一般写作活动，包括创建，取消链接和 mkdir 等系统调用;

(3) 将文件系统元数据，NAT，SIT 和 SSA 写入磁盘上的专用区域;

(4) 最后，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记录包之间的最新稳定包;
- NAT 和 SIT 位图指示包含当前包的一组 NAT 和 SIT 块;
- NAT 和 SIT 日志包含少量最近修改的 NAT 和 SIT 条目，以避免频繁的 NAT 和 SIT 更新;
- 活动段摘要块包含将来刷新到 SSA 区域的内存中 SSA 块;
- 孤立块保持“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并且一个进程删除它时会发生这种情况)，它应该被注册为一个独立的 inode，以便 F2FS 在突然关机后可以恢复它。

表 2: 实验平台信息

Target	System	Storage Devices
Mobile	CPU:Exynos 5410 Memory:2GB OS:Linux 3.4.5 Android:JB 4.2.2	eMMC 16GB: 2GB partition (114,72,12,12)
Server	CPU:Intel i7-3770 Memory:4GB OS:Linux 3.14 Ubuntu 12.10 server	SATA SSD 250GB: (486,471,40,140) PCIe (NVMe) SSD 960GB: (1295,922,41,254)

2.7.1 回滚恢复

突然关机后，F2FS 回滚到最新的一致检查点。为了在创建新包的同时保持至少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，F2FS 认为它是有效的。否则，它被丢弃。

同样，F2FS 也管理两套 NAT 和 SIT 块，由每个检查点包中的 NAT 和 SIT 位图来区分。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 在检查包中实现了一个 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 checkpoint 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，它会截断所有由它们引用的数据块，并最终释放孤立的 inode。然后，在前滚恢复过程成功完成之后，F2FS 将启动文件系统服务，并使用其位图所引用的一组一致的 NAT 和 SIT 块，这个将在下面描述。

2.7.2 前滚恢复

像数据库 (例如，SQLite) 这样的应用程序经常将小数据写入文件并保证持久性。一种支持 fsync 的最佳方法是触发检查点设置并使用回滚模型恢复数据。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文件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来增强 fsync 的性能。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选择性地查找数据块，F2FS 保留了一个特殊的标志直接节点块。

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，同时构建它们节点信息的列表。不参考自上一次更新后的块的数量检查点。(2) 通过使用列表中的节点信息，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。

3 实验部分

3.1 实验设置

我们评估 F2FS 在两个广泛分类的目标系统，移动系统和服务器系统。我们采用 GalaxyS4 智能手机来代表移动系统，并为服务器系统提供 x86 平台。表 2 总结了平台的规格。

对于目标系统，我们将 F2FS 从 3.15-rc1 主线内核分别移植到 3.4.5 和 3.14 内核。在移动系统中，F2FS 运行在最先进的 eMMC 存储上。在服务器系统的情况下，我们利用 SATA SSD 和 (更高速)PCIe SSD。请注意，每个存储设备下方表示的括号中的值表示以 MB/s 为单位的基本顺序读取/写入和随机读取/写入带宽。我们通过一个简单的单线程应用程序来测量带宽，该应用程序通过 O_DIRECT 触发 512KB 顺序 I/O 和 4KB 随机 I/O。

我们比较 F2FS 与 EXT4，BTRFS 和 NILFS2。EXT4 是一个广泛使用的更新就地文件系统。BTRFS 是写时复制文件系统，NILFS2 是 LFS。

表 3 总结了我们的基准测试和它们在生成 I/O 模式，触摸文件数量和最大大小，工作线程数量，读写比率 (R/W) 以及是否存在虚拟系统调用。对于移动系统，我们执行并显示 iozone 的结果，研究基本的文件 I/O 性能。由于移动系统受频繁 fsynccalls 代价高昂的随机写入，我们 runmobibench，一个宏基准，来衡量 SQLite 的性能。我们还在实际使用场景下重播了从“Facebook”和“Twitter”应用程序 (每个都被称为“Facebook-app”和“Twitter-app”) 收集的两个系统调用跟踪。

对于服务器工作负载，我们使用了一个称为 Filebench 的综合基准。它模拟各种文件系统工作负载，并允许快速直观的系统性能评估。我们在基准视频服务器，文件服务器，varmail 和 oltp 中使用了四个预定义的工作负载。它们在 I / O 模式和 fsync 使用方面有所不同。

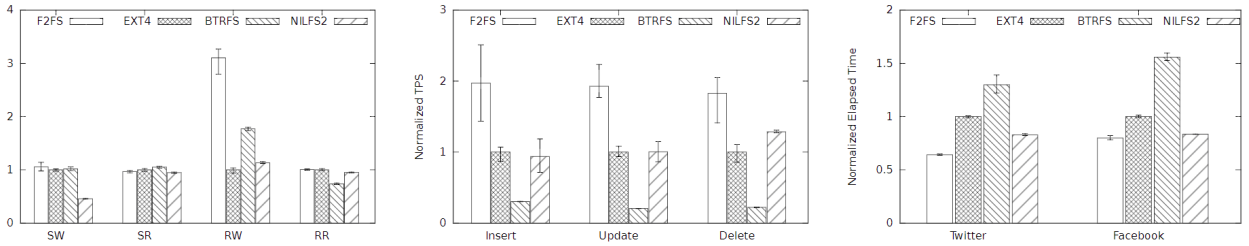
视频服务器主要是顺序读取和写入。文件服务器预先分配 80,000 个具有 128KB 数据的文件，随后启动 50 个线程，每个线程随机创建和删除文件以及读取和附加小数据到随机选择的文件。因此，这个工作负载表示具有许多大文件的缓冲随机写入和 nofsync 的情况。Varmail 使用 fsync 创建和删除一些小文件，而 oltp 预先分配十个大文件，并且用 200 个并行的线程随机更新他们的数据。

3.2 实验结果

本节给出了从深度块跟踪级分析中获得的性能结果和见解。我们对各种 I / O 模式 (即读取，写入，fsync 和丢弃 3)，I / O 数量和请求大小分布进行了预测。为了直观和一致的比较，我们将性能结果标准化为 EXT4 性能。我们注意到性能主要取决于顺序 I / O 和随机 I / O 之间的速度差距。在计算能力低，存储速度慢的移动系统中，I / O 模式及其数量是主要的性能因素。对于服务器系统来说，CPU 效率随着指令执行开销和锁定争用成为额外的关键因素。

表 3: 测试结果

Target	Name	Workload	Files	File size	Threads	R/W	fsync
Mobile	iozone	顺序和随机读写	1	1G	1	50/50	N
	SQLite	随机写和顺序同步	2	3.3MB	1	0/100	Y
	Facebook-app	随机写和顺序同步	579	852KB	1	1/99	Y
	Twitter-app	系统调用生成	177	3.3MB	1	1/99	Y
Server	videoserver	大部分顺序读写	64	1GB	48	20/80	N
	fileserver	许多大文件随机写	80,000	128KB	50	70/30	N
	varmail	许多小文件随机同步	8,000	16KB	16	50/50	Y
	oltp	大文件随机写和同步	10	800MB	211	1/99	Y



(a) iozone (b) SQLite (c) Facebook-app and Twitter-app

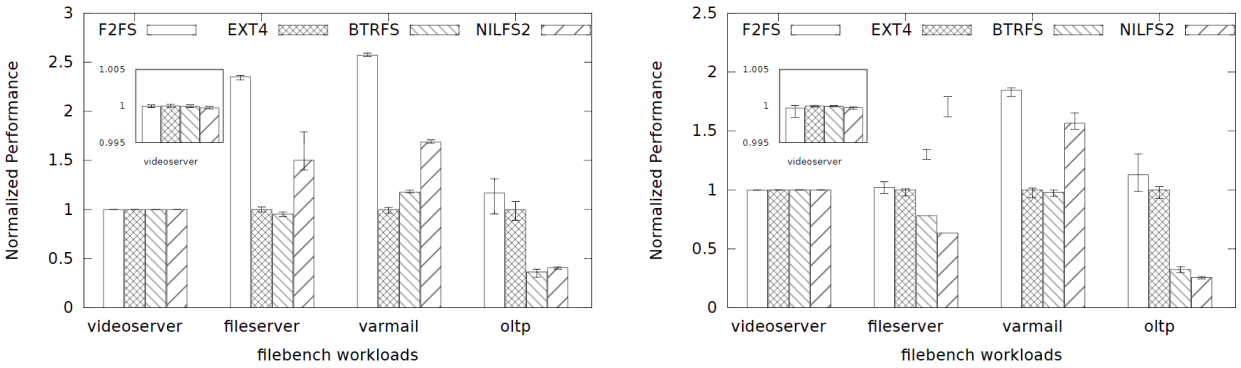
图 3: 在移动系统上的性能

3.2.1 在移动系统上的性能

图 3(a) 显示单个 1GB 文件的连续读/写 (SR / SW) 和随机读/写 (RR / RW) 带宽的 iozone 结果。在 SW 情况下，根据自己的数据刷新策略，NILFS2 比 EXT4 显示性能下降近 50%，因为它触发了昂贵的同步写入操作。在 RW 情况下，F2FS 比 EXT4 执行 3.1 倍，因为它将 4KB 随机写入的 90%转换为 512KB 顺序写入 (未直接显示在图中)。BTRFS 也能很好地执行 (1.8 倍)，因为它通过写时复制策略产生顺序写入。虽然 NILFS2 将随机写入转换为顺序写入，但由于成本高昂的同步写入，只能获得 10%的改善。而且，与其他文件系统相比，它可以发送多达 30%的写请求。对于 RR，所有文件系统都显示出相当的性能。由于树索引开销，BTRFS 显示性能略低。

图 3(b) 给出了以每秒事务数 (TPS) 衡量的 SQLite 性能，相对于 EXT4 的性能进行了归一化。在提前写入日志 (WAL) 日志模式下，我们测量了由 1000 条记录组成的数据库的三种类型的事务 - 插入，更新和删除。这种日志模式在 SQLite 中被认为是最快的。与其他文件系统相比，F2FS 的性能显着提高，性能比 EXT4 高出 2 倍。对于此工作负载，F2FS 的前滚恢复策略产生巨大的收益。事实上，在所有的案例中，F2FS 的数据写入量比 EXT4 减少了大约 46%。由于大量的索引开销，BTRFS 写入的数据比 EXT4 多出 3 倍，导致性能下降近 80%。与 EXT4 相比，NILFS2 具有相似的性能，数据写入量几乎相同。

图 3(c) 显示了完成播放 Facebook 应用程序和 Twitter 应用程序跟踪的标准化使用时间。他们使用 SQLite 来存储数据，与 EXT4 相比，F2FS 将经过的时间缩短了 20%(Facebook-app) 和 40%(Twitter-app)。



(a) SATA SSD (b) PCIe SSD

图 4: 在服务器系统上的性能

3.2.2 在服务器系统上的性能

图 4绘出了使用 SATA 和 PCIe SSD 研究的文件系统的性能。每个条指示正常化的性能 (即，如果条具有大于 1 的值，则性能改善)。

视频服务器主要生成连续的读取和写入操作，所有结果 (不论使用何种设备) 都不会在所研究的文件系统中产生性能差距。这表明 F2FS 对于正常的顺序 I / O 没有性能回归。

文件服务器有不同的 I / O 模式; 图 5比较从 SATA SSD 上的所有文件系统获取的块跟踪。仔细检查发现，由 EXT4 生成的所有写请求中只有 0.9%是 512KB，而 F2FS 是 6.9%(未直接显示在图中)。另一个发现是 EXT4 发出了许多小丢弃命令，并导致可见的命令处理，尤其是在 SATA 驱动器上; 它修剪数据写入覆盖的所有块地址的三分之二，接近 60%的丢弃命令是针对小于 256KB 的地址空间。相比之下，只有在触发检查点时，F2FS 才会丢弃以分段为单位的过时空间; 它可以减少块地址空间的 38%，而且不会丢弃任何小命令。这些差异导致了 2.4 倍的性能提升 (图 4(a))。

另一方面，BTRFS 将性能降低了 8%，因为它仅在所有写请求的 3.8%中发出 512KB 的数据写入。另外，如图 5(c) 所示，在读取服务期间，使用小丢弃命令 (对应于所有丢弃命令的 75%) 来修剪块地址空间的 47‰ 在 NILFS2 的情况下，多达 78%的写请求是 512KB(图 5(d))。但是，其周期性同步数据冲刷将 EXT4 的性能增益限制为 1.8 倍。在 PCIe 固态硬盘上，所有文件系统的运行方式都差不多。这是因为研究中使用的 PCIe SSD 可以很好地执行并行缓冲写入。

在 varmail 的应用中，F2FS 比 EXT4 在 SATA SSD 上的性能提高了 2.5 倍，在 PCIe SSD 上的性能提高了 1.8 倍。由于 varmail 使用并发 fsync 生成很多小写操作，结果再次强调了 F2FS 中 fsync 处理的效率。在 PCIe SSD 上，BTRFS 的性能与 EXT4 和 NILFS2 相当。

oltp 工作负载在一个 800MB 的数据库文件上产生了大量的随机写和 fsynccalls(与 varmail 不同，后者涉及很多小文件)。F2FS 显示性能优势比 SATA SSD 上的 EXT4-16%和 PCIe SSD 上的 13%显着。另一方面，BTRFS 和 NILFS2 在 PCIe 驱动器上的表现相当差劲。PCIe 驱动器上的快速命令处理和高效的随机写入看起来会改变性能瓶颈点，而 BTRFS 和 NILFS2 不会显示出强大的性能。

到目前为止，我们的研究结果清楚地表明了 F2FS 的总体设计和实施的相对有效性。现在我们将检查 F2FS 日志记录和清理策略的影响。

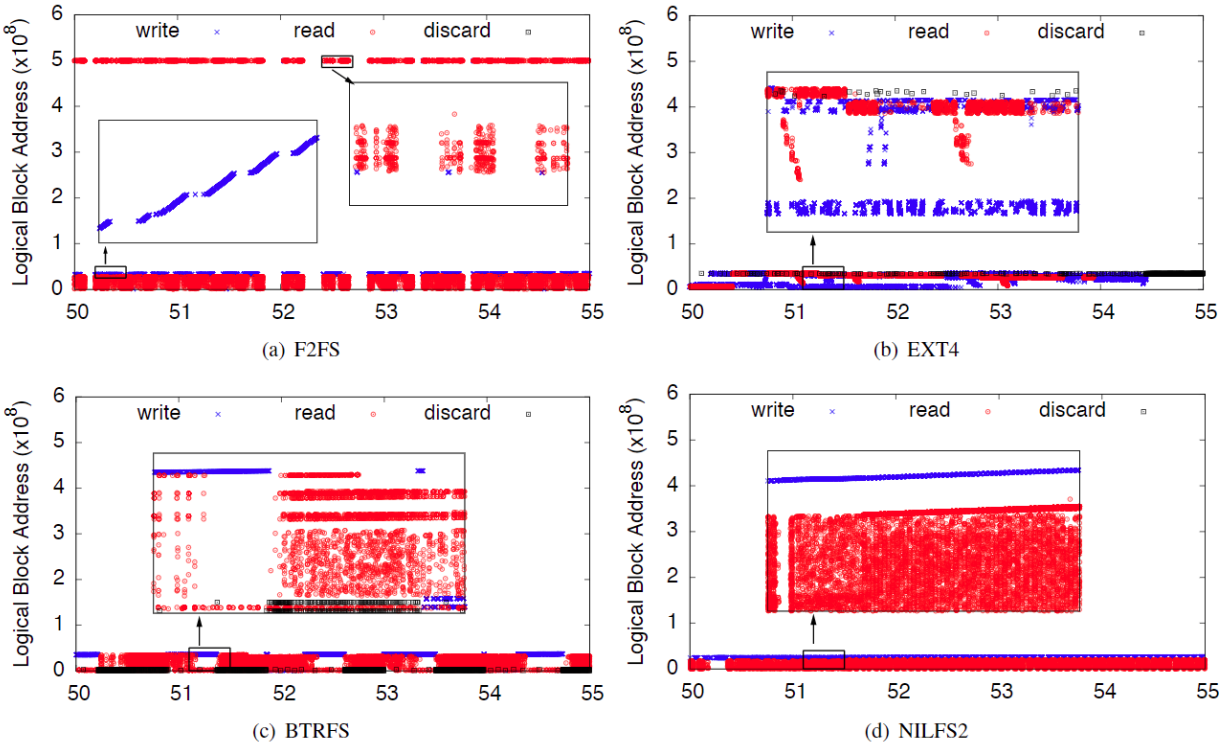


图 5: 根据时间的文件服务器工作负载的块跟踪

3.2.3 多头记录影响

本节研究 F2FS 多头记录的有效性。我们没有提供涵盖许多不同工作负载的广泛的评估结果，而是集中在一个捕捉我们设计的直觉的实验上。本节中使用的度量是在清理之前给定脏段中有效块的数量。如果冷热数据分离完全完成，则脏段将具有零有效块或段中有效块的最大数量 (在默认配置下为 512)。如果存储在段中的所有 (热) 数据已经失效，则老化的脏段将携带零个有效块。相比之下，一个充满有效块的脏段可能会保持冷数据。

在我们的实验中，我们同时运行两个工作负载：varmail 和 jpeg 文件的拷贝。Varmail 在 100 个目录中总共使用了 10,000 个文件，并写入了 6.5GB 的数据。我们复制每个大约 500KB 的 5,000 个 JPEG 文件，从而产生 2.5GB 的数据。请注意，F2FS 将 jpeg 文件静态分类为冷数据。在这些工作负载完成之后，我们计算所有脏段中有效块的数量。我们重复实验，因为我们将日志的数量从两个改为六个。

图 6给出了结果。对于两个日志，超过 75%的所有分段具有超过 256 个有效分块，而具有 512 个有效分块的“完整分段”非常少。由于双日志配置仅分割数据段 (所有脏段的 85%，未显示) 和节点段 (15%)，所以多头记录的有效性相当有限。添加两个更多的日志会稍微改变图片; 它增加了少于 256 个有效块的段的数量。它也略微增加了几乎完整的部分数量。

最后，有六个日志，我们清楚地看到冷热数据分离的好处; 具有零有效块的无空闲段的数量和完整段的数量显着增加。此外，有更多有效区块 (128 或更少) 的区段和更多有效区块 (384 或更多) 的区段。这种双峰分布的明显影响是提高了清洁效率 (因为清洁成本取决于受害者区段中有效区块的数量)。

在我们结束本节之前，我们进行了一些观察。首先，结果表明，更多的日志，允许更好地分离数据温度，通常会带来更多的好处。然而，在我们进行的特定实验中，四个日志对两个日志的好处是微不足道的。如果我们将热数据和热数据 (如表 1中所定义的) 中的冷数据分开，而不是来自冷数据 (缺省) 的热数据，则结果

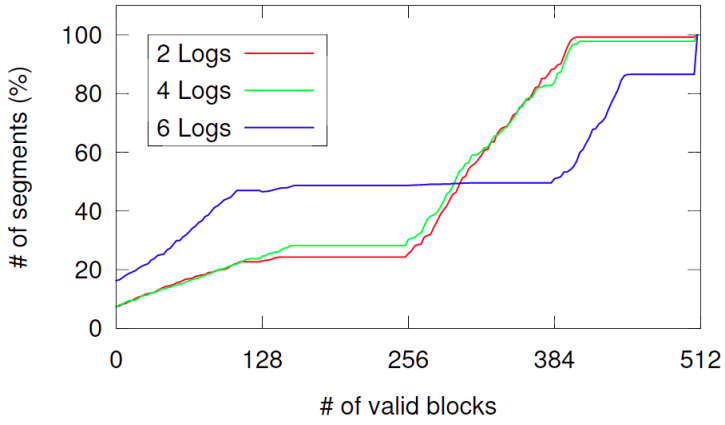


图 6: 根据段中有效块的数量的脏段分布

看起来不同。其次，由于脏段中的有效块的数量将随着时间的推移而逐渐减少，因此图 6中曲线的最左侧的膝部将向上移动 (根据所选择的记录配置以不同的速度)。因此，如果我们老化文件系统，我们预计多头测井优势将变得更加明显。充分研究这些观察是本文的范围。

3.2.4 清理成本

我们在这一部分量化 F2FS 清洗的影响。为了专注于文件系统级的清理成本，我们通过故意在 SSD 中留出足够的可用空间来确保 SSD 级别的 GC 不会在实际中出现。为此，我们格式化一个 250GB 的 SSD，并获得 (仅)120GB 的分区。

剩余 5%用于超额配置 (2.5 节) 后，我们将剩余容量划分为“冷”和“热”地区。我们构建了四个配置，通过填充两个区域来重新反映不同的文件系统利用率水平：80%(60(冷)：20(热))，90%(60:30)，95%(60:35) 和 97.5%(60：37.5)。然后，我们迭代 10 次运行的实验，每次运行随机将 4GB 的 20GB 数据写入热点区域。

图 7绘制了两个流程中的前十个运行结果：性能 (吞吐量) 和写入放大因子 (WAF)。4 他们是相对于在干净的 SSD 上获得的结果。我们做两个主要的观察。首先，较高的文件系统利用率导致 WAF 较大，性能下降。在 80%时，性能下降和 WAF 增加相当小。在第三次运行中，文件系统耗尽了空闲部分，并且性能下降。在这个运行期间，它从正常的日志记录切换到线程日志记录，结果，性能稳定。(我们在 3.2.5 节重新讨论了自适应线程化日志记录的效果。) 第三次运行之后，几乎所有的数据都是通过线程化的日志写入的。在这种情况下，不需要清理数据，而是记录节点。当我们把利用率从 80%提高到 97.5%时，GC 的数量增加了，而且性能的降低也变得更加明显。在 97.5%时，性能损失约为 30%，WAF 为 1.02。

第二个观察结果是，F2FS 在高利用率下不会显着增加 WAF; 自适应日志记录对于保持 WAF 是非常重要的。请注意，线程化日志记录会导致随机写入，而正常日志记录会发生顺序写入尽管随机写入相对昂贵，并且在许多文件系统中作为一种优选的操作模式激励随机记录，但是我们的设计选择 (切换到线程化记录) 是合理的，因为：当文件系统是碎片化，SSD 具有高随机写入性能。本节的结果表明，F2FS 成功地控制了高利用率的清洗成本。

显示背景清洁的积极影响并不简单，因为后台清洁在繁忙时期受到抑制。当我们在运行 10 分钟或更长的时间内插入空闲时间时，我们测量的性能提高了 10%，利用率达到了 90%。

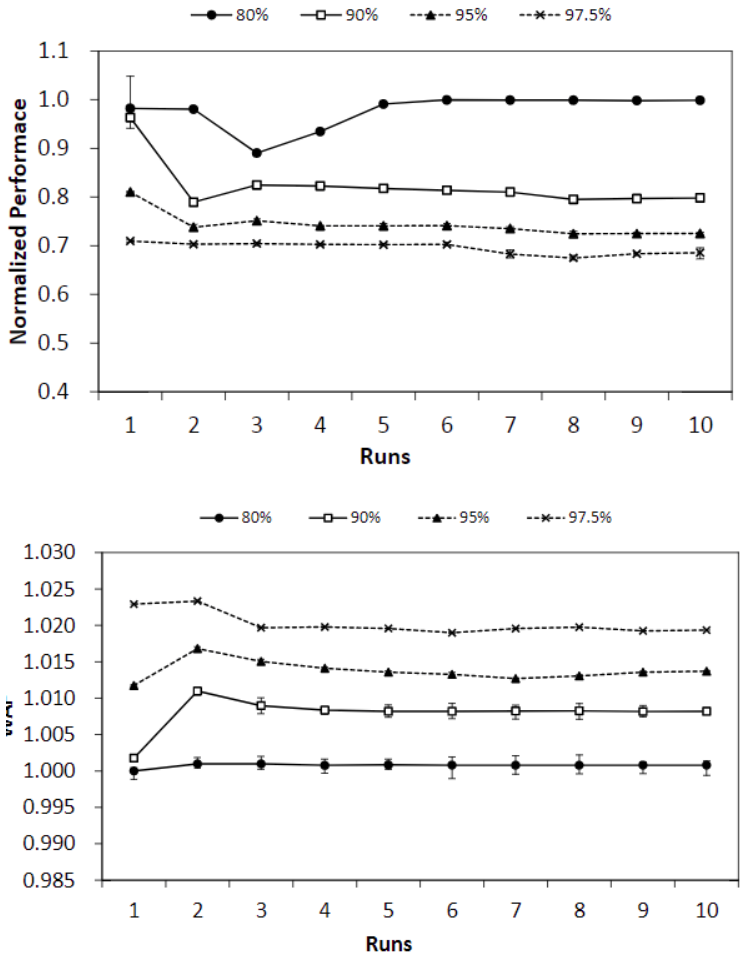


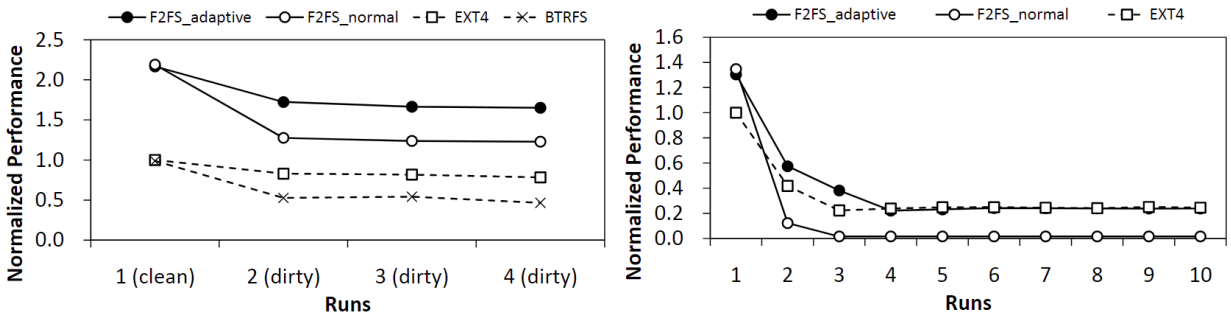
图 7: 前十次运行的相对性能 (上) 和写放大因子 (下)。四行分别记录了不同文件系统利用率级别的结果。

3.2.5 自适应日志性能

本节将深入探讨以下问题：使用线程化日志记录的 F2FS 自适应日志记录策略的有效性如何？默认情况下，当空闲部分的数量低于总部分的 5%时，F2FS 切换到正常日志记录的线程化日志记录。我们比较这个默认配置（“F2FS 自适应”）和“F2FS 正常”，它始终坚持正常的日志记录策略。对于实验，我们在 SATA SSD 上设计和执行以下两个直观的测试。

文件服务器测试。 这个测试首先填充目标存储分区 94%，数百个 1GB 文件。测试然后运行文件服务器工作负载四次，并测量性能趋势 (图 8(a))。在我们重复实验的时候，底层的闪存存储设备以及文件系统被分割了。因此，工作量的表现应该下降。请注意，我们无法使用 NILFS2 执行此测试，因为它以“无空间”错误报告停止。

EXT4 在第一轮和第二轮之间表现出最温和的表现，达到了 17%。相比之下，BTRFS 和 F2FS(特别是 F2FS 正常) 的性能下降了 22%和 48%，因为它们没有找到足够的连续空间。另一方面，F2FS 自适应为线程记录的总写入量的 51%(未在图中显示) 成功地将第二轮中的性能下降限制到 22%(与 BTRFS 相当并且距离 EXT4 不太远)。结果，F2FS 全面维持了 EXT4 以上的两倍以上的性能改善率。所有的文件系统都显示出在第二轮以后的性能。



(a) fileserver (random operations) on a device filled up 94%.

(b) iozone (random updates) on a device filled up 100%.

图 8: 文件系统老化最差情况下性能下降率

进一步的检查表明，由于前台清理，F2FS 正常写入比 F2FS 自适应多 27%的数据。BTRFS 的大量性能是由于大量使用小丢弃命令造成的。

iozone 测试。 该测试首先创建 16 个 4GB 文件和额外的 1GB 文件，直到填充设备容量 (　100%)。然后运行 iozone 在 16 个 4GB 文件上执行 4KB 随机写入。总写入量为每个文件 512MB。我们重复这个步骤十次，结果是相当苛刻的，因为 BTRFS 和 NILFS2 都没有完成“没有空间”的错误。请注意，从理论上看，EXT4 是一个更新就地文件系统，在这个测试中表现最好，因为 EXT4 发出随机写入而不创建额外的文件系统元数据。另一方面，像 F2FS 这样的日志结构的文件系统可能会承受很高的清理成本。另请注意，此工作负载会将存储设备中的数据分段，并且由于工作负载触发重复的设备内部 GC 操作，存储性能将受到影响。

在 EXT4 下，性能下降约为 75%(图 8(b))。在正常情况下，F2FS 性能下降到一个非常低的水平 (第三轮的 EXT4 低于 5%)，因为文件系统和存储设备都在忙于清理碎片容量，以回收新的记录空间。显示 F2FS 自适应更适合处理这种情况; 它在前几轮 (当碎片不严重时) 表现比 EXT4 更好，并且表现出与 EXT4 非常类似的性能，因为随着写入更随机的进行。

本节中的两个实验表明，自适应日志记录对于 F2FS 在高存储利用率下保持其性能至关重要。自适应日志记录策略还可以有效地限制由于碎片导致的 F2FS 的性能下降。

4 相关工作

本节讨论三种类型的与我们有关的以前的工作-日志结构文件系统，闪存文件系统以及 FTL 特殊优化。

4.1 日志结构文件系统 (LFS)

从 Rosenblum 等人最初的 LFS 提案开始，已经在日志结构文件系统 (HDD) 上做了大量的工作。威尔克斯等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。Matthews 等人提出了一个适应性的清洁政策，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。还有研究表明线程化日志在高度利用的体积中提供了更好的性能。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。Wang 和 Hu 提出区分缓冲区缓存中的活动和非活动数据，而不是将它们写入单个日志，并在清理过程中将其分开。他们通过监视访问模式来确定哪些数据是活动的。Hylog 采用混合方法; 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洁成本。

SFS 是基于 NILFS2 实现的 SSD 的文件系统。和 F2FS 一样，SFS 使用日志来消除随机写入。为了降低清洁成本，他们根据跟踪写入次数和每块年龄测量的“更新可能性”(或热度)，将缓存中的热数据和冷数据分开，如。他们使用迭代量化来基于测量的热度将片段分成组。

与使用访问模式的运行时监视的热/冷数据分离方法不同，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。虽然我们的实验结果表明，我们采取的简单方法是相当有效的，但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。

NVMFS 是一个实验文件系统，假设两种不同的存储介质：NVRAM 和 NAND 闪存 SSD。NVRAM 的快速字节寻址存储容量用于存储热和元数据。另外，写入 SSD 的顺序与 F2FS 一样。

4.2 闪存文件系统

针对使用原始 NAND 闪存作为存储器的嵌入式系统，已经提出并实现了许多文件系统。这些文件系统可以直接访问 NAND 闪存，同时还可以修补所有芯片级别的问题，如损耗均衡和坏块管理。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。

Josephson 等人提出了直接文件系统 (DFS)，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。

4.3 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。DAC 提供了页面映射 FTL，通过在运行时监视访问，基于更新频率对数据进行聚类。为了减少大页面映射表的开销，DFTL 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。

5 总结和展望

F2FS 是为现代闪存存储设备设计的全面的 Linux 文件系统，目前已经在业界广泛采用，比较有名的就是国内的华为手机，使用了这个技术，并宣称手机可以 3 年不卡。本文介绍了 F2FS 的关键设计和实现细节。其评估结果强调了 F2FS 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。未来还可以在 F2FS 的基础上不断添加新的优化和新的功能，使得这个文件系统更加强大和稳定。

参考文献

[1] Lee, Changman, et al. ”F2FS: A New File System for Flash Storage.” FAST. 2015.

[2] Lee, Sang-Won, et al. ”A log buffer-based flash translation layer using fully-associative sector translation.” ACM Transactions on Embedded Computing Systems (TECS) 6.3 (2007): 18.

[3] Kim, Jesung, et al. ”A space-efficient flash translation layer for CompactFlash systems.” IEEE Transactions on Consumer Electronics 48.2 (2002): 366-375.