学号 2017282110204

密级

海量存储技术课程论文

F2FS：面向闪存存储的文件系统

院（系）名 称 ： 计算机学院

专 业 名 称 ： 计算机技术

学 生 姓 名 ： 曹珪琰

指 导 教 师 ： 何水兵 副教授

二〇一七年十二月

摘　要

NAND 闪存设备广泛应用于各种移动设备，如智能手机，平板电脑和 MP3 播放器。此外，服务器系统开始使用闪存设备作为其主存储器。闪存尽管有着广泛的应用，但也有一些限制，如擦除前写请求，擦写块顺序写入和每个擦除块有限的写入周期。

为了在闪存存储设备上有更佳的性能表现，F2FS[1]（Flash Friendly File System，闪存友好的文件系统）作为一种 Linux 文件系统被设计及实现。该文件系统的关键性设计是充分利用闪存的特性，进行性能的优化，特别针对 NAND 闪存存储介质做了友好设计。

F2FS 使用基于日志文件系统（log-structured file system，LFS）方案，并使之更加适应新的存储介质 NAND。同时，修复了旧式日志结构文件系统的一些已知问题，如“漫游树”（wandering tree）的滚雪球效应和高“清理”开销。根据内部几何结构和闪存管理机制，闪存存储设备有很多不同的属性，所以 F2FS 的设计者增加了多种参数，不仅用于配置磁盘布局，还可以选择分配和清理算法，优化并行 I/O，提高性能。

原文作者介绍了 F2FS 的主要设计思想，数据结构，算法和性能。实验结果展示了 F2FS 突出的性能。在最新的移动终端系统和服务器系统上，性能相比于 EXT4 系统均有大幅度的提升。

关键词：文件系统、NAND 闪存、F2FS

目　录

1 绪论 4
1.1 研究背景 4
1.2 本文研究内容和章节安排 5
2 F2FS 文件系统的设计与实现 6
2.1 闪存友好的磁盘布局 6
2.2 文件结构和目录结构 8
2.2.1 文件索引树对 F2FS 的意义 9
2.2.2 NAT 解决 wandering Tree 问题 10
2.3 F2FS 的块分配 10
2.4 空闲空间管理 11
2.5 Cleaning 过程 12
2.6 文件系统空间写满的处理 12
3 实验分析 14
3.1 实验安排 14
3.2 实验结果 14
3.2.1 移动端和服务器端的实验结果 14
4 总结与展望 17
参考文献 19

1 绪论

1.1 研究背景

NAND 闪存广泛应用于各种移动设备，如智能手机，平板电脑和 MP3 播放器。此外，服务器系统开始使用闪存设备作为其主存储器。 闪存尽管有着广泛的应用，但也有一些限制，如擦除前写请求，擦写块顺序写入和每个擦除块有限的写入周期。

早期，许多消费电子设备直接利用“裸”的 NAND 闪存存储平台。然而，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被称为 FTL（flash translation layer，闪存转换层），它解决了 NAND 闪存的限制，并提供了一个通用的解决方案。这种闪存存储解决方案的例子包括 eMMC（embedded multimedia card，嵌入式多媒体卡），UFS（universal flash storage，通用闪存）和 SSD（solid-state drive，固态硬盘）。 通常，这些新一代的闪存存储设备相比于硬盘驱动器，它们的机械计数器部件显现出低得多的访问延迟。例如随机 I/O，SSD 比 HDD 要快上几个数量级。

然而，在一定的闪存存储设备的使用条件下，NAND 闪存的缺点就体现出来了。例如，Min 等人[2]观察到，频繁地随机写入 SSD 会导致底层介质（underlying media）内部碎片化，并降低 SSD 性能。研究表明，随机写入模式相对于移动设备上资源受限的闪存解决方案更为常见。Kim 等人[3]观察到，移动设备中 I/O 请求的 80％以上是随机的，超过 70％的随机写入通过触发 fsync，如 Facebook 和 Twitter。这种特定的 I/O 模式是因为这些应用程序中的 SQLite[4]。除非经过仔细处理，否则频繁的随机写入和闪存操作可能会增加闪存设备的 I/O 延迟并严重损害设备使用寿命。

随机写入的有害影响可以通过日志文件系统（LFS）的方法来减少[5]。 例如，预期文件系统，如 Btrfs[6]在 NAND 闪存 SSD 上表现良好，然后其没有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欠佳。我们认为传统的硬盘驱动器文件系统设计策略（尽管有性能的提升）不能充分利用和最优化 NAND 闪存介质的使用。

原文作者主要介绍了针对闪存存储设备优化的新文件系统——F2FS 的设计和实现。作者对 F2FS 设计的主要贡献：

（1） 设计和实现了新的文件系统 F2FS，以优化 NAND 闪存的性能。

（2） 与 Linux 文件系统(Ext4, Btrfs, Nilfs 2)，进行了性能的比较。

1.2 本文研究内容和章节安排

本文的主要内容，下一节主要描述了 F2FS 的设计和实现。第 3 章节主要内容是实验结果和讨论。

2 F2FS 文件系统的设计与实现

F2FS[1]建立在日志文件系统（LFS）的基础之上，但与 LFS 已经大不相同，可以将 F2FS 作为一个新的 Linux 文件系统，作者将其与两个最流行的 Linux 文件系统 EXT4 和 BTRFS 进行比较。作者还考虑了两大系统平台，即移动终端系统（安卓系统）和服务器系统，测试了 F2FS 的性能。

F2FS 的设计主要创新点可以分为以下方面：

（1）闪存友好的磁盘布局：F2FS 采用三种基本的管理单元：段（segment），节（section）和区域（zone）。对多个单独的区域以段为基本单元分配存储块，以节为单元执行“清理”（cleaning）操作。引入这些基本单元是为了与底层的闪存转换层（FTL）的单位保持一致，以避免不必要的数据复制与转换。

（2）文件索引结构：基于日志的文件系统（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，其间接索引块也应被更新。这样的递归更新导致了一连串的写入，造成了“漫游树”（wandering tree）问题[7]。原作者提出了一个新的索引表，称为节点地址表（node address table），解决这个问题。

（3）F2FS 的块分配：原作者设计了在日志生成期间（即块分配时间）热/冷数据（hot/cold data）分离处理的方案。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各自独立的日志段。由于闪存介质运行的并行性，多个活动段可以同时运行而不需要频繁的管理操作。F2FS 建立了所谓“仅追加日志记录”（append only logging），将随机写入转换为顺序写入。在高存储利用率的情况下，日志记录策略使用线程化记录（threaded logging）[8]以避免长时间的写入延迟。实际上，线程化日志记录将新数据写入脏段（dirty segment）中的空闲空间，可以并行清理。这种策略在闪存设备上运行良好。

2.1 闪存友好的磁盘布局

F2FS 将整个卷切分成大量的段（Segments），每个段的大小是固定的 2 MB。连续的若干个段构成节（Section），连续的若干个节构成区域（Zone）。如图 1 所示，F2FS 将整个卷分成固定大小的段，每个段对应 Segment Number，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。这些单元在记录和清理过程中非常重要，在第 2.4 节和第 2.5 节将进一步讨论。

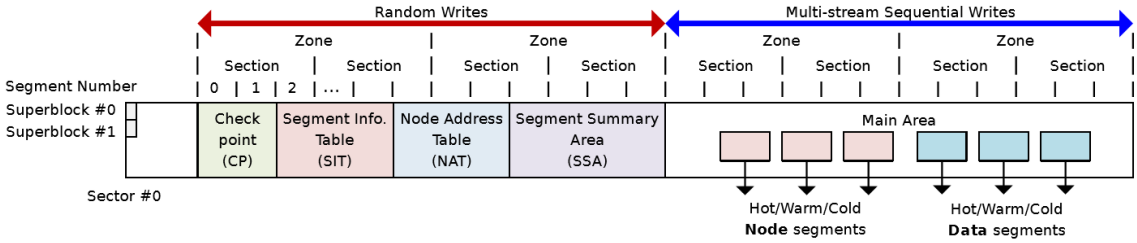


图 1

F2FS 将整个卷分成六个大的区域（areas）：

（1）超级块（Super Block，SB）：具有基本的分区信息和 F2FS 的默认参数，它在格式化时间给出，并且不可更改。

（2）检查点（Check Point，CP）：用于保持和恢复文件系统状态，主要用于在突然断电事件发生后发现时间点。

（3）段信息表（Segment Information Table，SIT）：包含段的基本信息，如有效块的数量和位于主区域（main area）中所有块的有效性位图。

（4）节点地址表（Node Address Table，NAT）是定位存储在主区域中的所有“节点块”的块地址表。存储在 Main 区域的所有节点（inode 和 索引节点）数据块的块地址表。所有的节点(node)块由 NAT 映射，这意味着每个 node 的位置由 NAT 表来转换。考虑 wandering tree 问题，F2FS 使用这种节点映射索引方式可以切断由于叶子节点修改操作引起的节点更新传播问题。

（5）段摘要区（Segment Summary Area，SSA）存储表示主区域中所有块的所有者信息的摘要条目，例如父索引节点号和其父节点偏移量。

（6）主区域（Main area）大小是 4KB 块。每个块被分配和输入为节点或数据。 节点块包含索引节点或索引的数据块，而数据块包含目录或用户文件。值得注意的是，一个块不会同时存储数据和节点块。

具体磁盘布局见图 2 所示。

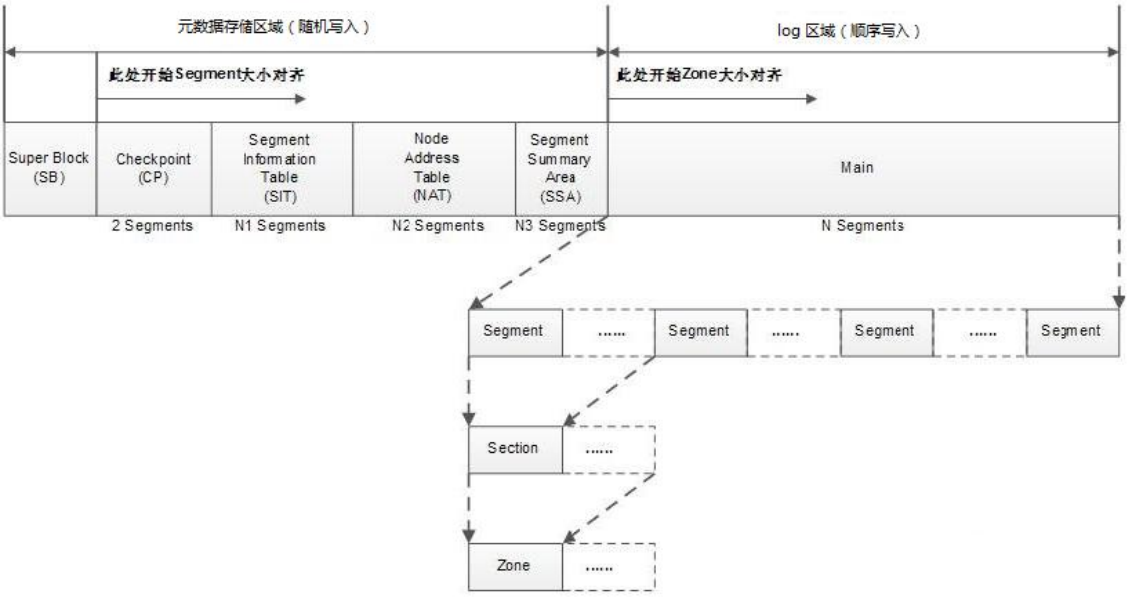


图 2

F2FS 查询操作：假设一个文件所在目录是“/dir/file”，F2FS 执行以下步骤：

（1）从 NAT 获得的含有根节点位置的块；

（2）在根索引节点块中，从其数据块查找目录名 dir，并获取其索引节点号;

（3）通过 NAT 将检索到的索引节点（inode）号转换为物理位置;

（4）通过读取物理位置相应的块得到名为 dir 的 inode;

（5）在 dir inode 中，它能确定找到 file 的文件名，最后通过重复文件步骤（3）和（4）获得文件索引节点。真实数据是从主区域上取回，并通过相应的文件结构获得索引。

2.2 文件结构和目录结构

日志文件系统引入的 inode 映射将 inode 号转换为磁盘上的位置。相比之下，F2FS 使用索引节点结构，来扩展 inode 映射，定位更多的索引块。每个节点块都有一个唯一的节点 ID。通过使用节点 ID 作为索引，NAT 用于记录所有节点块的物理位置。

节点块有三种类型：inode，直接节点 direct node 和间接节点 indirect node。inode 块包含文件的元数据，如文件名，inode 号，文件大小，atime 和 dtime。而直接节点块包含数据的块地址，间接节点具有定位另一节点块的节点 ID。

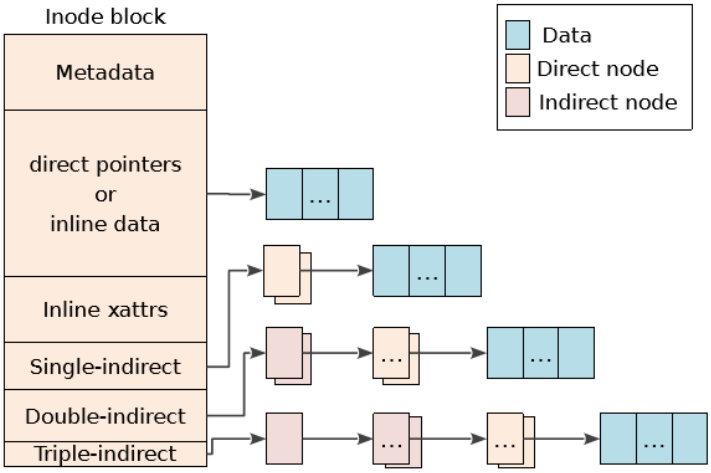


图 3

如图 3 所示，F2FS 使用指向直接和间接节点块的文件索引来消除更新传播，即“漫游树”问题[7]。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 更新一个指针块（总共三个），而 F2FS 仍然只更新一个。 inode 块包含文件数据块的直接指针，两个单一间接指针，两个双重间接指针和一个三重间接指针。 F2FS 支持行数据和内联扩展属性，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。

2.2.1 文件索引树对 F2FS 的意义

大多数现代文件系统都喜欢使用 B-trees 或类似的结构来管理索引（index）以定位文件中的 块。大多数文件系统中通过使用 extents 来减少文件数据块的总索引大小。F2FS 不采用 B-tree 结构管理索引，也不使用 extents 减少文件数据块索引的大小（虽然 F2FS 为每个节点(node)维护一个 extent 作为提示）

F2FS 使用索引树（如原始 Unix 文件系统和 Ext3 的索引树）索引数据块。但是这种索引树机制有一定的代价——这也是其他文件系统抛弃这种索引方式的主要原因——但是这种方式对 LFS 却有好处。因为 F2FS 不使用 extents，任意给定文件的索引树是固定的而且大小已知。这意味着，当数据块由于 cleaning 操作需要转移数据时，不可能因为可用 extents 发生改变而导致索引树变得更大（原来文件中每个 extent 很大，一个文件的索引树仅仅用少数几 extents 就可完全表示，而新的存储部分每个可用 extent 都很小，要存储原来的文件就需要更多的 extents，

导致文件的索引树变大）的情况产生——这可能会导致一种尴尬的情况产生就是cleaning 的目的本来是要释放空间，而现在索引树变大却导致空间变小了。LFS 文件系统也因为相同的原因，使用了许多相同的编排处理方式。

显然，所有这些需求都要求 F2FS 的 inode 比 Ext3 的 inode 要大许多。因为Copy-on-write 不适用于小于块大小的对象，因而 F2FS 中每个 inode 占用的空间大小是一个 4 KB 的块，该 4 KB 的空间中提供了大量的空间用于索引。甚至提供空间存储（基本的）文件名，或者名字之一，以及其父节点的 inode 号，这简化了在文件系统 crash 恢复过程中近期创建的文件的恢复操作，减少了为保证文件安全需要写入的数据块的数量。

2.2.2 NAT 解决 wandering Tree 问题

任何基于 Copy-on-write 技术的文件系统的一个缺点在于，无论何时要改写一个数据块，该数据块的地址就会因为数据重写到其他位置而发生改变。然后，在索引树中该块的父节点也因此必须重写到其他位置，如此往复下去直到重写操作传播到索引树的根。LFS 的 logging 特性意味着在恢复过程中向前回滚可以重建近期对索引树的修改，因而所有这些修改不必立即写到磁盘（只是写到 log 中），但是这些修改最终还是会写到磁盘的，这就将更多的工作丢给 cleaner 去做。

这是 F2FS 为了方便而使用其底层 FTL 去处理的另一个区域。“元数据”区域("meta" area)中的内容是一个 NAT——Node Address Table（节点地址表）。这里的节点是指 inode 和间接索引块，以及用于存储扩展属性的数据块。当 inode 的地址存储在目录中，或者索引块存储在一个 inode 中或者其他的索引块中的时候，它存储的不是（存储的内容所在的）地址，而是其在 NAT 中的偏移。实际的（存储内容的）块地址是存储在 NAT 中对应的偏移处。这意味着当一个数据块被写入，仍然需要更新和修改指向该数据块的节点，但是修改该节点仅需更新对应的 NAT 项。NAT 是元数据的一部分，NAT 使用两个位置(two-location)的 Journaling（依靠 FTL 实现写聚合），因而不需要进一步的索引。

2.3 F2FS 的块分配

LFS 有一个主要的日志区域，但是 F2FS 保持六个以最大化冷热数据分离的效

果。对于节点和数据块，有三个温度级别：热，温和冷。可以调整写入流的数量（例如，通过将冷和热组合成一个流），根据数据所属的类型，利用对应的 log 分配的空间将数据写入该空间，这样做可以在给定的存储设备和平台上提供更好的结果。

如表 1 所示。直接节点块被认为比间接节点块更热，因为它们的更新频率更高。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。

满足以下三个条件之一的数据块被认为是冷的：

清理后的数据块。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。为 F2FS 支持扩展的属性操作。

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识别它们。

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数据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，则每个日志段对应于表 1 中的温度。在日志中，F2FS 将每个节点和数据类型中的冷日志和温日志结合在一起。

表 1

Type	Temp.	Objects
Node	Hot	Direct node blocks for directories
	Warm	Direct node blocks for regular files
	Cold	Indirect node blocks
Data	Hot	Directory entry blocks
	Warm	Data blocks made by users
	Cold	Data blocks moved by cleaning; Cold data blocks specified by users; Multimedia file data

2.4 空闲空间管理

LFS 有两种机制用于空闲空间管理：Threaded log 和 copy-and-compaction。Copy-and-compaction 机制就是 log-structured 文件系统中常常提到的 cleaning 操作，Copy-and-compaction 机制非常适用于具有非常好的连续写性能的设备，但是该机制的 cleaning 操作开销对性能影响很大。相反，Threaded log 机制的弊端是要承受随机写，但是不需要 cleaning 过程。F2FS 采用混合机制，其中 Copy-and-compaction 机制是默认采取的机制。但是 F2FS 会根据文件系统的状态(在没有足够的 clean 的段)动态地更改为 Threaded log 机制。

为了使 F2FS 与底层的 flash 存储设备对齐，F2FS 在以 section 为单位的存储

单元中分配一个 segment，F2FS 期望 section 的大小与 FTL 中的垃圾回收单元的大小一致。此外，至于 FTL 中的映射粒度，F2FS 尽可能从不同的 Zone 中分配每个（上述 6 个）活跃的 log 的 section，因为 FTL 可以将活跃的 log 中的数据根据其映射粒度（并行）写到分配的单元中。

2.5 Cleaning 过程

F2FS 可以在需要的时候(on-demand)或者空闲的时候以后台处理的方式进行 clean 操作。On-demand cleaning 操作是由于没有足够的空闲段服务 VFS 调用而触发开启的 cleaning 操作，后台 cleaner 由内核线程操作，当系统 I/O Idle 的时候触发 cleaning 操作。

F2FS 支持两种选择待清理段的策略：贪心和成本效能算法。在贪心算法中，F2FS 选择具有最少有效数据块个数的段；在成本效能算法中，F2FS 根据段的年龄以及有效数据个数选择段，以解决贪心算法中的 log block 颠簸问题(频繁的 cleaning 操作，频繁的数据块迁移，极端情况，选中的段可能在很长一段时间内一直是贪心算法的选择结果)。F2FS 对 on-demand cleaner 采用贪心算法选择待清理的段，而对于后台 cleaner 采用成本效能算法。为识别出选中的段中哪些数据有效，F2FS 管理一个位图，每一位表示一个数据块的有效性，该位图描述了 Main Area 中所有数据块的数据有效性。

2.6 文件系统空间写满的处理

传统文件系统中如果没有剩余空间，直接返回错误就行。而对于 Log-structured 文件系统则没有这么潇洒地简单返回错误的处理机制，因为没有空闲块并不表示已使用的块上的数据都有效，在 clean 操作后又会释放出空闲块。通常一个比较有意义的做法是为 Log-structured 文件系统提供过量的空间，使得总是有空闲的 Section 用于 cleaning 操作中的有效数据的拷贝存储。

FTL 恰好利用这种方法，提供过量空间用于“cleaning”以及用于替换高度损耗造成的坏块。FTL 在内部处理过量空间，因而当 F2FS 开始用完空间的时候，它本质上放弃了整个 log-structured 的理念，只是随机地在其可以写入的地方写入数据。但此时，inode 和索引块仍然会仔细认真对待（不是随意写），有少量的过量空间用于它们的写入。但是对于数据只能采取本地更新，或是写入任意能找到的空

闲块中。这样，可以预见的是当 F2FS 变满的时候，性能会下降，这与大多数的文件系统表现一样。

3 实验分析

3.1 实验安排

我们在两个广泛应用的系统平台上，即移动终端系统和服务器系统上评估F2FS。我们采用三星 Galaxy S4 智能手机来代表移动系统，为服务器系统提供 x86 平台。总结如表 2 所示。

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: *(1,295, 922, 41, 254)*

表 2 平台规格

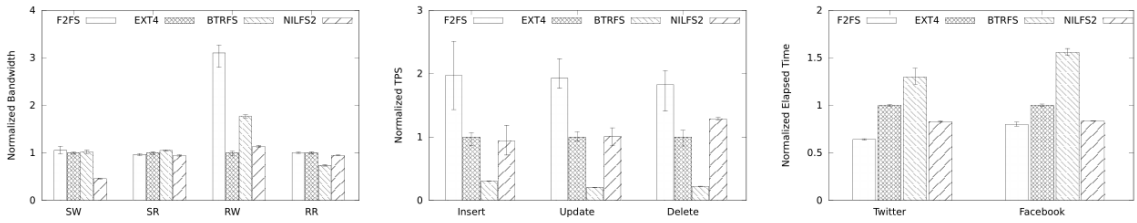
Target	Name	Workload	Files	File size	Threads	R/W	`fsync`
Mobile	iozone	Sequential and random read/write	1	1G	1	50/50	N
	SQLite	Random writes with frequent `fsync`	2	3.3MB	1	0/100	Y
	Facebook-app	Random writes with frequent `fsync`	579	852KB	1	1/99	Y
	Twitter-app	generated by the given system call traces	177	3.3MB	1	1/99	Y
Server	videoserver	Mostly sequential reads and writes	64	1GB	48	20/80	N
	fileserver	Many large files with random writes	80,000	128KB	50	70/30	N
	varmail	Many small files with frequent `fsync`	8,000	16KB	16	50/50	Y
	oltp	Large files with random writes and `fsync`	10	800MB	211	1/99	Y

表 3 实验设置

表 3 根据 I/O 模式，触摸文件的数量和最大大小，工作线程的数量，读写的比率（R/W）以及是否存在 fsync 系统，总结了我们的 benchmarks 和它们的特性。 对于移动系统，我们执行和显示 iozone 的结果，研究基本的文件/操作。 由于移动系统需要经常性的 fsync 调用来进行昂贵的随机写入，所以 mobibench [9]可以测量 SQLite 的性能。我们还在实际使用场景下从“Facebook”和“Twitter”应用程序收集的两个系统调用跟踪。

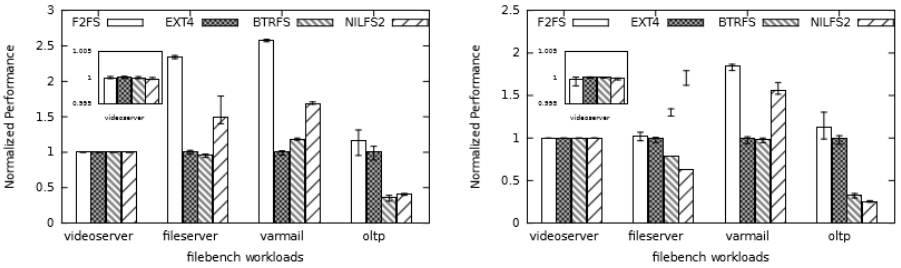
3.2 实验结果

3.2.1 移动端和服务器端的实验结果



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

图 4



(a) SATA SSD (b) PCIe SSD

图 5

图 4（a）显示了单个 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，根据自己的数据流策略，NILFS2 的性能比 EXT4 降低了近 50%，因为它会定期触发昂贵的同步写操作。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。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，它的写请求量高达 30%。对于 RR，所有的文件系统都表现出相当的性能。由于树索引开销，BTRFS 显示性能略低。

窗体顶端

图 4（b）给出了以每秒事务数（transactions per second，TPS）衡量的 SQLite 性能，相对于 EXT4 的性能进行了归一化。我们衡量三种类型的事务：插入，更新和删除；在提前写入日志（WAL）日志模式下由一个由 1,000 条记录组成的数据库。这种日志模式在 SQLite 中是最快的。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。对于这个工作量，F2FS 的前滚恢复策略产生巨大的好处。在所有的案例中，F2FS 的数据写入量比 EXT4 减少了大约 46%。由于大量索引开销，BTRFS 写入 3 倍以上的 EXT4，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 4（c）显示了两个应用程序使

用 SQLite 存储数据，与 EXT4 相比，F2FS 的使用时间缩短了 20％（Facebook-app）和 40％（Twitter-app）。

图 5 展示了使用 SATA 和 PCIe 固态盘的研究文件系统的性能。 每一柱状值表示标准化的性能，即如果柱状值具有大于 1 的值表示性能提升。视频服务器主要生成连续的读取和写入操作，所有的结果，不管使用什么设备，都没有显示研究文件系统之间的性能差距。 这表明 F2FS 对于顺序 I / O 没有性能的降低。

原作者对移动端和服务器端做了进一步的实验与分析，篇幅有限，更多的实验请查看原文[1]。

4 总结与展望

经过阅读文献和网上浏览资料，总结如下，主要论述了 F2FS 文件系统的设计背景和它解决的主要问题：

- LFS（Log structured File System）

为管理磁盘上的大的连续的空间以便快速写入数据，将 log 切分成 Segments，使用 Segment Cleaner 从重度碎片化的 Segment 中转移出有效的信息，然后将该 Segment 清理干净用于后续写入数据。

- Wandering Tree 问题

在 LFS 中，当文件的数据块被更新的时候是写到 log 的末尾，该数据块的直接指针也因为数据位置的改变而更改，然后间接指针块也因为直接指针块的更新而更新。按照这种方式，上层的索引结构，如 inode、inode map 以及 checkpoint block 也会递归地更新。这就是所谓的 wandering tree 问题。为了提高性能，数据块更新的时候应该尽可能地消除或减少 wandering tree 的更新节点传播。

F2FS 解决方案

（1）用节点（node）表示 inode 以及各种索引指针块；

（2）引入节点地址表(NAT) 包含所有“node”块的位置，这将切断数据块更新的递归传播。

- 清理（cleaning）开销

为了产生新的 log 空闲空间，LFS 需要回收无效的数据块，释放出空闲空间，这一过程称为 cleaning 过程。cleaning 过程包含如下操作：

（1）通过查找 segment 使用情况链表，选择一个 segment；

（2）通过 segment summary block 识别出选中的 segment 中的所有数据块的父索引结构并载入到内存；

（3）检查数据及其父索引结构的交叉引用；

（4）选择有效数据进行转移有效数据。

cleaning 工作可能会引起难以预料的长延时，因而 LFS 解决的一个重要问题是对用户隐藏这种延时，并且应当减少需要转移的数据的总量以及快速转移数据。

F2FS 解决方案（最小化 Cleaning 开销）：

（1）支持后台 Cleaning 进程；

（2）支持贪心和成本效能(转移数据最少)的待清理 section 选择算法；

（3）支持 multi-head logs 用于 hot/cold 数据分离；

（4）引入自适应 logging 用于有效块分配。

参考文献

[1] Lee C, Sim D, Hwang J Y, et al. F2FS: a new file system for flash storage[C]// Usenix Conference on File and Storage Technologies. USENIX Association, 2015:273-286.

[2] 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.

[3] H. Kim, N. Agrawal, and C. Ungureanu. Revisiting storage for smart phones. ACM Transactions on Storage (TOS), 8(4):14, 2012.

[4] Using databases in android: SQLite. http: //developer.android.com/guide/ topics/data/data-storage.html#db.

[5] 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.

[6] O.Rodeh,J.Bacik,andC.Mason. Btrfs: Thelinux b-tree filesystem. ACM Transactions on Storage (TOS), 9(3):9, 2013.

[7] A. B. Bityutskiy. JFFS3 design issues. http:// www.linux-mtd.infradead.org, 2005.

[8] 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.

[9] S. Jeong, K. Lee, S. Lee, S. Son, and Y. Won. I/O stack optimizationfor smartphones. In Proceedings of the USENIX Anual Technical Conference (ATC), pages 309 - 320, 2013.