|  |  |
| --- | --- |
| 学号 | 2017202110075 |
| 密级 |  |

**海量存储技术课结课论文**

|  |
| --- |
| **B*e*trFS:一个右优化的写优化文件系统** |

|  |  |
| --- | --- |
| 院（系）名 称 ： | 计算机学院 |
| 专 业 名 称 ： | 计算机应用技术 |
| 学 生 姓 名 ： | 邹建文 |
| 学 生 学 号 ： | 2017202110075 |
| 指 导 教 师 ： | 何水兵 副教授 |

二〇一七年十二月

摘要

*Bɛ-tree*文件系统或BetrFS是第一个使用写优化索引的内核文件系统。写入优化索引（WOI）是有希望的存储系统构建模块，因为它们可以有效地实现微粒和大扫描。

以前有关基于WOI的文件系统的工作已经显示出了希望，但也受到了本文所述的几个公开问题的阻碍。例如，FUSE向文件系统发出许多查询，将读密集型工作负载叠加在写密钥上，从而降低了WOI的有效性。转向内核实现可以通过提供更好的读取控制来解决这个问题。本文还提供了几种实现技术来利用内核基础架构，而不会限制写入性能。

我们的研究结果表明，BetrFS为任意微数据操作（包括创建小文件，更新元数据和小写入大文件或小文件以及大型顺序I/O）提供了良好的性能。在一个微数据基准测试中，BetrFS提供了4倍以上的ext4或XFS性能.BetrFS是一个正在进行的原型工作，需要额外的数据结构调整来匹配当前通用文件系统的某些操作，如删除，目录重命名和大 顺序写入。尽管如此，许多应用程序实现了显着的性能改进。例如，Linux内核源码的就地rsync实现了比其他商品文件系统大约1.6-22倍的加速。

关键字：*Bɛ-tree；*BetrFS；FUSE；WOI

目录

[1 绪论 1](#_Toc500611882)

[2 动机 3](#_Toc500611883)

[2.1 microwrite 问题 3](#_Toc500611884)

[3 B*e*trFS 设计 5](#_Toc500611885)

[4 BetrFS文件系统架构 6](#_Toc500611886)

[4.1 BetrFS数据和元数据索引 6](#_Toc500611887)

[4.2实现BetrFS操作 7](#_Toc500611888)

[5 系统设计中的写优化 10](#_Toc500611889)

[5.1 无污染缓存 10](#_Toc500611890)

[5.2 写优化FUSE 11](#_Toc500611891)

[5.3 Ext4 作为一个块管理器 11](#_Toc500611892)

[6 实现 13](#_Toc500611893)

[6.1 移植方法 13](#_Toc500611894)

[6.2 klibc框架 14](#_Toc500611895)

[6.3 TokuDB的变化 15](#_Toc500611896)

[6.4 未来工作和局限性 15](#_Toc500611897)

[7 评估 17](#_Toc500611898)

[7.1 Microwrites 17](#_Toc500611899)

[7.2 大型读写 19](#_Toc500611900)

[7.3 目录操作 20](#_Toc500611901)

[7.4系统调用Nanobenchmarks 20](#_Toc500611902)

[7.5 空间开销 21](#_Toc500611903)

[8 结论 22](#_Toc500611904)

[参考文献 23](#_Toc500611905)

# 1 绪论

当今的应用程序呈现出各种各样的I/O模式，使得通用文件系统的性能调整成为令人沮丧的平衡行为。某些软件（如病毒扫描和备份）需要大量的顺序扫描数据。其他软件需要很多小写（微粒）。例如电子邮件传送，为编辑应用程序创建锁定文件，对较大文件进行较小更新或更新文件的一次。根本的问题在于，文件系统设计者的工具箱中的许多标准数据结构以牺牲另一个为代价优化一个案例。

写入优化指标（WOI）[4,8-10,23,27,28]的最新进展令人兴奋，因为它们有可能实现有效的微粒和大扫描。最好的关键优势是它们可以在B树的目标和距离查询性能匹配或提高的同时，将数据的采集速度提高到B树的两个数量级[4,9]。

WOI在商业键值存储和数据库方面已经取得了成功[2,3,12,17,20,33,34]，而且之前关于文件系统中的WI的研究已经显示出了希望[15,25,31]。然而，在生产质量优化的文件系统方面的进展受到了几个公开挑战的阻碍，我们在这篇文章中提到：

•代码复杂性。一个生产质量的WI很容易就是5万行复杂的代码，这很难在操作系统内核中发挥作用。先前的WOI文件系统已经在用户空间中实现。

•FUSE浪费了微波炉性能。几乎每次更新之前，FUSE[16]都会向底层文件系统发出查询，将搜索密集型工作负载叠加在写密集型工作负载上。虽然WOI对于点查询来说不比任何其他明智的数据结构更差，但是写操作比读取要快得多，注入不必要的点查询可以使写优化的优点无效。

•将文件系统抽象映射到WOI上。我们不能通过简单地放弃WOI来替代B-tree来实现写优化的全部潜在性能优势。内核基础架构的架构和使用必须利用新数据结构的性能优势。

本文介绍了*Bɛ-tree*文件系统，或BetrFS，这是第一个内核文件系统，旨在充分利用写入优化的优势。具体来说，BetrFS是使用成熟的，设计良好的*Bɛ-tree*从Tokutek的分形树索引实现，称为TokuDB[33]。我们的设计是针对分形树索引的性能特点进行量身定制的，但是除此之外，我们的设计决定可能适用于其他写入优化的文件系统。

实验表明，BetrFS可以放弃对文件创建，大文件随机写入，递归目录遍历（例如在查找，递归greps，备份和病毒扫描中）的性能的数量级改进，以及元数据 更新（例如每次读取文件时更新文件）。

本文的贡献是：

•内核写优化文件系统的设计和实现。

•确保本地和快速写入的模式，用于将VFS操作映射到写优化索引。

•使用未经修改的操作系统内核基础架构（为传统文件系统设计）的设计，尽可能减少对写入优化的影响。例如，BetrFS使用操作系统内核高速缓存来加速读取，而不会限制比磁盘扇区小的写入操作。

•全面评估BetrFS原型的性能。例如，我们的结果显示，在一个小写的microbenchmark和rsync等应用程序上，速度提高了两个数量级。

我们的结果表明，一个精心设计的*Bɛ-tree*基于树的文件系统在几乎每个操作上都可以匹配或优于传统的文件系统，有些是一个数量级。与最先进的文件系统比较，比如ext4，XFS，zfs和btrfs支持这种说法。我们认为，少数几个较慢的操作不是WOI的基础，但是可以在未来的工作中结合算法的进步和工程的努力加以解决。

# 2 动机

本节总结了关于microwrite重要性日益增加的背景，解释了WOI如何工作。

## 2.1 microwrite 问题

microwrite是一种写入操作，其中建立时间（即在常规磁盘上的寻道时间）超过数据传送时间。传统的文件系统数据结构迫使文件系统设计者在为有效微粒进行优化和高效扫描之间进行选择。

原地更新文件系统[11,32]通过将相关项目（例如目录中的条目或文件中的连续块）保持在彼此附近来针对扫描进行优化。但是，由于项目更新到位，更新性能往往受到底层磁盘随机写入延迟的限制。

基于B树的文件系统存储在B树中逻辑相邻的相关项目，但是B树不能保证逻辑上相邻的项目在物理上相邻。随着B树年龄的增长，由于插入的节点分裂和删除的节点合并，树叶散落在磁盘上。在老龄化的B树中，叶子的逻辑顺序和物理顺序之间几乎没有关系，而阅读新叶子的代价涉及数据传输成本和搜索成本。 如果叶子太小而无法分摊搜索成本，则范围查询可能会很慢。寻找成本可以通过使用较大的叶子来摊销，但是这进一步限制了更新性能。

在另一个极端，日志文件系统[5,7,26,29,30]优化写入。日志记录确保文件的创建和更新速度很快，但是生成的数据和元数据可能在整个日志中传播，导致从磁盘读取数据或元数据时性能较差。在大型扫描（递归目录遍历和备份）中，这些性能问题尤其值得注意，这些扫描无法通过缓存加速。

microwrite的瓶颈造成了一系列应用的问题。高性能计算检查点系统生成如此多的微粒，因此定制的文件系统PLFS被设计为通过利用检查点工作负载的细节来有效地处理它们[5]。电子邮件服务器通常很难管理大量小消息和关于这些消息的元数据，如读标志。桌面环境将首选项和活动状态存储在键值存储区（即注册表）中，以便访问和更新密钥不需要文件系统级微数据操作。Unix和Linux系统管理员通常通过禁用atime选项来报告10-20％的性能提升[14]。保持正确的时间行为会导致沉重的微波炉负荷，但是一些应用需要精确的时间值。

即使存储系统使用SSD，Microwrites也会导致性能问题。在基于B树的文件系统中，小写入触发整个B树节点的较大写入，这可以进一步放大到SSD上的整个擦除块。在一个日志结构的文件系统中，微粒可以引起沉重的清理活动，特别是当磁盘快满时。无论是哪种情况，额外的写入操作都会缩短SSD的使用寿命，并会浪费带宽来限制性能。

# 3 B*e*trFS 设计

BetrFS是一个内核文件系统，旨在充分利用*Bɛ-tree*的性能优势。整个系统架构如图3.1所示。

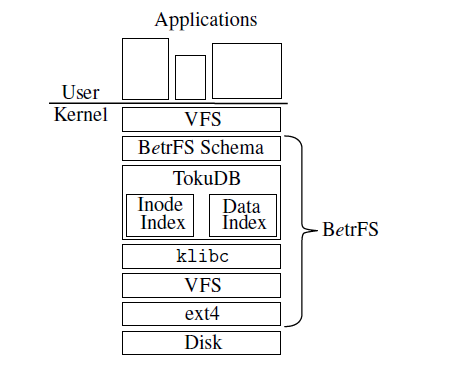


图3.1 BetrFS体系结构

BetrFS VFS模式尽可能地将文件系统操作转换为高效的*Bɛ-tree*操作。从*Bɛ-tree*中获得良好性能的关键是（1）使用upsert操作来更新文件系统状态，（2）组织数据以便文件系统扫描可以实现为*Bɛ-tree*中的范围查询。我们在第4节描述了我们的模式如何实现这些目标。

通过实现BetrFS作为内核文件系统，我们避免了FUSE的性能开销，这对于写优化的文件系统来说尤其有害。我们还公开了优化文件系统与内核页面缓存交互的机会。

BetrFS的堆叠文件系统设计清楚地分离了实现写优化索引的复杂任务，从块分配和空闲空间管理。TokuDB的内核端口将数据存储在底层的ext4文件系统中，但任何文件系统都应该足够。

将一个45KLoC数据库移植到内核中是一项非常重要的任务。 我们通过编写一个shim层（我们称之为klibc）将TokuDB移植到内核中，该层将TokuDB外部依赖转换为锁定，内存分配和文件I/O的内核函数。第6部分介绍了klibc，总结了我们从项目中吸取的经验和教训。

# 4 BetrFS文件系统架构

*Bɛ-tree*实现了一个键值存储，因此BetrFS必须将文件系统操作转换为键值操作。本节介绍用于执行此翻译的BetrFS模式，并说明该模式如何利用*Bɛ-tree*的性能优势。

## 4.1 BetrFS数据和元数据索引

BetrFS使用底层数据库中的两个索引来存储文件系统数据和元数据：元数据索引和数据索引。由于键和值都可以被变量化，所以BetrFS能够将许多索引条目打包到每个*Bɛ-tree*节点中。

**元数据索引。**BetrFS原型维护一个将完整路径名（相对于挂载点）映射到文件元数据（大致相当于struct stat内容）的索引：

path\_(size,owner,timestamps,etc . . .)

元数据索引旨在支持高效的文件创建，删除，查找和目录扫描。索引按照斜杠的数量排序，然后按字典顺序排列。因此，同一个目录中的项目被连续存储，如图4.1所示。按照这个顺序，可以递归或不递归地扫描一个目录，作为一个范围查询。

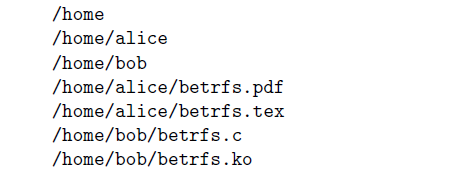


图4.1 在BetrFS中使用的排序顺序的示例

**数据索引。**虽然键和值可能是可变大小的，但BetrFS原型将文件分割成4096个字节的块，以便更好地与页面缓存集成。因此，数据索引映射（文件，偏移）元组到块：

(path,block-number)\_data[4096]

数据索引中的键也按字典顺序排序，这保证了文件内容在逻辑上相邻，因此，如2.2节所述，几乎总是物理上相邻的磁盘上。这使得文件内容可以以接近磁盘带宽的顺序被读取.BetrFS通过简单地省略数据索引中的稀疏块来实现稀疏文件。

BetrFS使用可变大小的值来避免零填充每个文件的最后一个块。这种优化避免了将缓冲区的未使用区域归零的CPU开销，然后在将节点写入磁盘之前将零压缩掉。对于小文件基准测试，这种优化大大降低了开销。例如，这种优化将TokuBench（§7）的吞吐量提高了50-70％。

## 4.2实现BetrFS操作

**赞成盲文写。**在很多文件系统代码中，潜在的假设是数据必须以磁盘扇区粒度写入。因此，一个小的写入操作必须首先将周围的磁盘块放入缓存中，修改该块，然后将其写回。这种模式反映在Linux页面缓存帮助函数function\_block\_write\_begin()中。B**etrFS**避免了这种读取修改-写入模式，而是在可能的情况下发出无读写的盲写入。

在BetrFS中读写文件。BetrFS使用数据索引中的范围查询实现文件读取. *Bɛ-tree*可以在有效的磁盘带宽下从磁盘加载大范围查询的结果。

*BetrFS*通过插入和插入支持任何大小的有效文件写入。应用程序写入小于一个4K块成为格式的消息：

UPSERT(WRITE*,*(*path,n*)*,offset,v,l),*

这意味着应用程序写入*l*字节数据v在指定文件的第n个块的给定偏移处。插入消息完全封装了块修改，不需要读取-修改-写入。整个块的写入是通过插入（也称为put）来实现的，该插入是块的替换，并且行为相似。

在跨越多个块的大写入的情况下，插入遵循从根到树叶复制数据的类似路径。 对于大型写入，*Bɛ-tree*实现有一些优化，可以跳过到叶节点的路径上的中间节点，但是对于大型顺序文件写入，它们不够积极，无法实现完整的磁盘带宽。我们将这个问题留作未来工作，因为解决方案必须谨慎处理与待处理消息和分裂叶节点相关的一些细微问题。

BetrFS中的文件系统元数据操作。如图4.2所示，BetrFS还将几乎所有元数据更新（如时间戳更改，文件创建和符号链接）转换为upserts。

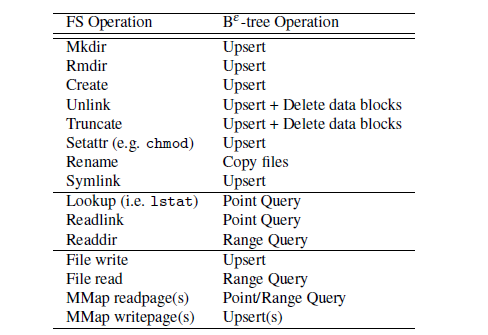


图4.2 基本文件系统操作的BetrFS实施策略。

BetrFS中唯一没有插入的元数据更新是取消链接，截断和重命名。我们现在解释执行这些操作的障碍，作为插件，我们为未来的工作留下。

取消链接和截断可以从文件中删除块。BetrFS以最简单的方式执行此操作：对要删除的块执行范围查询，并为数据索引中的每个块发出TokuDB删除。 虽然TokuDB删除操作是使用upserts实现的，但是发出O(n)upserts可能会使这个任务变得非常昂贵。

其次，通过完整路径进行键控使得递归目录遍历有效，但是实现高效重命名并不重要。例如，我们当前的实现通过重新插入所有键-值对，然后删除旧键来重命名文件和目录，从而有效地执行被重命名的文件或目录的深层副本。一个简单的解决方案是添加一个inode风格的间接层，第三个索引。这种方法是很好理解的，并且可以牺牲一些阅读地点随着树龄。我们认为，数据结构级别的优化可以提高重命名的性能，我们将为今后的工作留下。

尽管上述架构可以使用upsert来对文件系统进行大部分更改，但许多POSIX文件系统函数指定了在更改文件系统之前操作系统必须检查的先决条件。例如，在创建文件时，POSIX要求操作系统检查文件是否已经存在，并且用户对包含目录有写权限。在我们的实验中，文件和目录信息的操作系统缓存能够回答这些查询，使文件创建等能够以最高速度运行。

**崩溃一致性。**我们使用TokuDB事务机制来实现崩溃一致性。TokuDB事务相当于完整的数据日志记录，所有数据和元数据更新记录到底层ext4文件系统中的文件中。日志条目按顺序停用，并且在TokuDB日志记录机制之前没有更新应用于磁盘上的树。条目被附加到两个内存中的日志缓冲区之一（默认情况下为16 MB）。这些缓冲区每秒都会旋转并刷新到磁盘，或者缓冲区溢出。

尽管公开事务API是可能的，但是BetrFS目前仅使用事务作为内部一致性机制。 除了写数据（每个数据块使用一个事务）以外，BetrFS通常每个系统调用使用一个事务。在我们当前的实现中，元数据上的事务在执行时保持适当的VFS级别的互斥锁，使得事务冲突和异常终止异常罕见。

**压缩。**压缩对性能非常重要，特别是对于按键。两个索引都使用完整路径作为关键字，这些关键字可能很长且重复，但是TokuDB的压缩缓解了这些开销。 使用quicklz [24]，在我们的实验中排序的路径名称压缩20倍，使磁盘空间管理上空。

数据压缩的使用也意味着在读取文件系统级块和从磁盘读取块之间不存在一对一的对应关系。叶节点通常为4 MB，并且压缩可以将多于64个文件系统块打包到叶中。根据我们对大数据读写的经验，数据压缩可以提高文件系统吞吐量，比禁用压缩高达20％。

# 5 系统设计中的写优化

在设计BetrFS时，我们设定了在现有Linux VFS框架内工作的目标。一个根本的挑战是，在某些方面，支持代码假定读取与写入一样昂贵，并且是就地更新所必需的。写入优化的使用违反了这些假设，因为子块写入可能比读取更快。 本节介绍了我们在保留Linux支持基础架构的同时提高BetrFS性能的几种策略。

## 5.1 无污染缓存

BetrFS利用Linux页面缓存实现有效的小读取，避免一般的磁盘读取，并促进内存映射文件。默认情况下，当应用程序写入当前处于缓存中的页面时，Linux会将其标记为脏，并在稍后写出。这样，多个应用程序级写入页面可以吸收在缓存中，只需要一次写入磁盘。但是，在BetrFS中，小写操作非常便宜，以至于这种优化并不总是有意义的。

在BetrFS中，写入系统调用永远不会在缓存中弄脏干净的页面。当一个应用程序写入一个干净的缓存页面时，BetrFS向*Bɛ-tree*发出一个upsert，并将写入应用到该页面的缓存副本。因此，缓存的内容仍然与磁盘上的数据同步，并且缓存的页面保持清洁。

请注意，BetrFS的方法并不总是比吸收缓存写入和写回整个块更好。例如，如果应用程序对同一个块执行了数百次小型写入，则将页面标记为脏并且等到应用程序完成将最终内容写回磁盘会更有效。生产版本的BetrFS应该包括启发式检测这种情况。但是，我们发现在没有这种优化的情况下，我们原型的性能已经足够好了，所以我们还没有实现它。

缓存中的页面变脏的唯一情况是当文件被写入内存映射时。内存管理硬件不支持对内存映射文件的精细跟踪写操作 - 操作系统只知道内存页面中的内容已被修改。因此，BetrFS的mmap实现使用默认的读取和写入页面机制，以页面粒度运行。

我们的设计使页面缓存与磁盘保持一致。我们利用页面加速器实现更快的热缓存读取，但是避免了不必要的全页面回写。

## 5.2 写优化FUSE

我们将BetrFS作为内核文件系统实现，因为FUSE体系结构包含若干设计决策，可能会破坏写优化文件系统的潜在性能优势。当与用户空间文件系统进行通信时，FUSE具有额外的上下文切换和数据编组的知名开销。但是，由于完全不同的原因，FUSE对编写优化的文件系统造成了特别的损害。

FUSE可以将写入密集型转换为读取密集型工作负载，因为它在大多数文件系统更新之前（实际上是在其之后）向用户空间文件系统发出查询。例如，FUSE发出GETATTR调用（类似于调用stat（））以查找文件的整个路径，每次应用程序查找该文件。对于大多数内核文件系统，后续查找可以由 内核的目录缓存，但FUSE保守地认为底层文件系统可以异步改变（这可以是真实的，例如在网络文件系统中）。

这些搜索可以阻塞写入优化的数据结构，其中插入比搜索快两个数量级。 TokuFS作者明确指出这些搜索是FUSE实现令人失望的原因[15]。

TableFS作者确定了FUSE开销的另一个来源：内核中inode信息的双缓存[25]。这会降低缓存的有效命中率。对于慢速文件系统来说，额外缓存未命中的开销可能并不重要。对于写入密集型工作负载的写优化数据结构，高速缓存未命中的开销可能很大。

## 5.3 Ext4 作为一个块管理器

由于TokuDB将数据存储在可变大小的压缩节点中，因此TokuDB依赖底层文件系统充当磁盘的块和空闲空间管理器。传统的文件系统在磁盘上相邻地存储大文件块的做法很好，特别是在以大块执行文件写操作的时候。

我们坚持把这个设计放在TokuDB的内核端口上，而不是重新发明轮子。 BetrFS将树节点表示为底层文件系统上的一个或多个大文件中的块，在我们的原型中，它是未经修改的ext4，具有有序数据模式和直接IO。我们依靠ext4正确地发出磁盘写入缓存的障碍，尽管禁用磁盘的写入缓存不会显着影响我们工作负载的性能。换句话说，所有的BetrFS文件系统更新，数据或元数据通常以数据写入的形式出现，而底层ext4文件系统的fsync则确保了BetrFS日志写入的持久性。尽管层之间存在一些重复的工作，但是我们期望有序的日志模式将这种情况降到最低，因为典型的BetrFS实例从ext4的角度跨越了11个文件。也就是说，这些裁员可以在未来的工作中精简。

# 6 实现

我们将TokuDB移植到Linux内核中，作为获得写优化数据结构实现的最方便的方式，而不是从头开始在内核中实现写优化的数据结构。这种数据结构实现可能很复杂，特别是在 调整I/O和渐近合并行为。

在本节中，我们解释了如何将大部分TokuDB代码移植到内核中，我们在这个过程中面临的挑战，从经验中吸取的经验教训，以及将来的实现工作。图6.1总结了BetrFS中的代码行，包括将VFS层与TokuDB接口的代码，klibc代码，以及对Linux内核代码的小改动，下面将对此进行介绍。BetrFS原型使用Linux版本3.11.10。

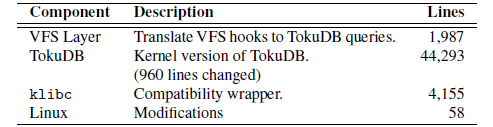


图6.1 BetrFS中的代码行，按组件分类

## 6.1 移植方法

我们最初决定移植是可行的，因为TokuDB只有很少的库需求，而且是用类似C的C ++编写的。换句话说，TokuDB使用的C ++特性主要是在编译时实现的（例如，名称修改和更好的类型检查），并且不需要运行时支持异常等功能。我们的方法应该适用于其他的工具，比如一个LSM树，因为它的实现遵循了类似的编码风格。

因此，我们很大程度上可以将用作二进制blob的TokuDB代码作为代码创建一个内核模块（.ko文件）。我们导出了BetrFS VFS层使用的接口使用C链接，TokuDB从klibc导入的类似的声明接口是C链接。

我们通常最小化对TokuDB代码的更改，并以对象文件粒度选择导入的代码。在少数情况下，我们添加了编译时宏以消除代码路径或函数，这些代码路径或函数不会被使用，但仍需要繁琐的依赖。最后，当一个特别繁琐的用户级API（例如fork）仅在少数几个地方使用时，我们重写了代码以使用更合适的内核API。我们调用由TokuDB klibc导入的结果集合。

## 6.2 klibc框架

图6.2总结了由klibc导出的ABI。在很多情况下，内核ABI是直接导出的，比如memcpy，或者是对诸如同步和内存分配等功能的直接包装。在少数情况下，变化更为复杂。

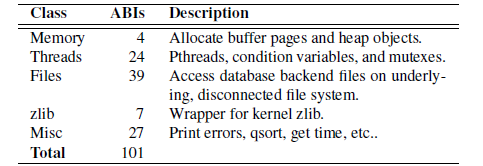


图6.2 由klibc导出的ABI函数的类

在TokuDB代码中使用errno提出了特别的挑战。Linux将错误代码作为负值返回，而libc只返回负值，并将错误代码放入每个线程变量errno中。检查一个负值，读取TokuDB中的errno是非常普遍的，改变错误处理行为是不切实际的。我们最终在Linux任务结构中添加了一个errno字段; 生产实现将会改为错误传递代码。

尽管在内核抽象中包装pthread抽象是非常简单的，静态初始化和直接访问pthread结构会产生问题。主要问题是将pthread抽象转换为内核抽象将pthread结构定义中的成员替换。静态初始化将不正确地初始化修改后的结构。一旦pthread结构的大小发生变化，我们不得不消除导入系统pthread头部的任何代码，以免嵌入实例的大小。

在将ext4作为块存储重用时，我们在创建模块级文件句柄和路径时面临一些挑战。文件句柄更直接：我们可以创建一个模块级句柄表，并使用pread（无游标）API ext4读取和写入。我们不得不修改Linux来导出几个直接接受结构文件的VFS帮助函数，而不是走进程级文件描述符表。我们还修改了ext4，以接受来自不是用户级地址的O\_DIRECT标志的读取输入。

当BetrFS在底层的ext4文件系统上分配，打开或删除一个块存储时，模块基本上被chroot成一个与主树断开的ext4文件系统。因为这是内核代码，我们也希望避免基于当前进程的凭据进行权限检查。因此，路径操作包括“上下文切换”操作，其中当前任务的文件系统根目录和证书被保存和恢复。

## 6.3 TokuDB的变化

除了少数例外，我们能够在内核中使用TokuDB，而不做大的修改。本小节概述了需要重构代码的问题。

我们遇到的第一个问题是TokuDB在整个过程中自由使用堆栈分配。一个函数在堆栈上分配了一个12KB的缓冲区！相比之下，Linux内核中的堆栈大小在编译时是固定的，默认为8KB。在大多数情况下，我们可以使用编译时警告来识别大堆栈分配并将其转换为堆分配并添加自由函数。在这些结构对性能至关重要的情况下，例如数据库游标，我们修改了TokuDB代码以使用更快的分配方法，例如内核缓存或每个CPU变量。同样，我们重写了几个递归函数来使用循环。尽管如此，我们发现仍然可以使用更多modestized帧的深层栈，并将堆栈大小增加到16 KB。我们计划在未来的工作中以最大的堆栈大小统治。

最后，我们发现futexes和需要更改代码的内核等待队列之间的小错配。 从本质上讲，pthread条件变量的最近实现不会因为不相关的中断而唤醒睡眠线程，使得用户空间中的安全（尽管可能不太明智）不会在从pthread cond wait返回之后再次检查不变量。Linux内部的等价物（如等待事件）可以在不重新检查不变量的情况下以难以区分的方式虚假地唤醒线程。因此，我们必须将所有的pthread cond等待调用放在一个循环中。

## 6.4 未来工作和局限性

BetrFS原型是一个持续的努力。这个努力已经达到了足够的成熟度，证明了在内核文件系统中写入优化的力量。但是，今后的工作有几点需要改进。

目前从TokuDB代码库中缺少的最有用的功能是范围upserts;upserts只能应用于单个密钥，或者广播到所有密钥。目前，文件删除必须通过为文件中的每个数据块创建一个删除upsert来实现;创建适用于有限范围的单个upsert的能力将是有用的，我们将其留作未来工作。支持这种抽象的主要困难在于调整upsert应该被冲刷到叶子上的积极性，还是应用于点需求的查询;我们将这个问题留作未来的工作。

组织磁盘放置的一个微妙的折衷是在重命名和搜索性能之间。BetrFS键盘文件的路径，当前导致重命名将文件从一个磁盘位置复制到另一个。这可以通过添加一个间接层（即inode号）来明显减轻。但是，这与在目录层次结构中保留数据局部性的目标不一致。我们计划研究用于保存局部性的更有效的目录操作的技术。同样，我们目前的原型不支持硬链接。

我们目前的原型还包括一些磁盘数据的双重缓存。几乎我们所有的实验都测量了冷藏缓存行为，所以这并不影响我们结果的保真度。profligate内存使用仍然是问题。从长远来看，我们打算更好地整合这些层，以及消除模拟的文件句柄和路径。

# 7 评估

我们围绕以下问题组织评估：

•BetrFS上的microwrites比其他通用文件系统更有效率吗？

•BetrFS上的大型读写是否至少与其他通用文件系统相竞争？

•其他文件系统操作如何在BetrFS上执行？

•BetrFS的空间（内存和磁盘）开销是多少？

•应用程序是否在BetrFS上实现更好的整体性能？

除非另有说明，否则基准测试是冷藏测试。 所有文件系统都可以从页面和目录缓存中获得同样的效果。我们有兴趣测量缓存未命中的效率。

所有的实验结果都在戴尔Optiplex 790上收集，该处理器采用4核3.40 GHz Intel Core i7 CPU，4 GB RAM和250 GB，7200 RPM ATA磁盘。每个文件系统使用一个4096字节的块大小。系统运行Ubuntu 13.10,64位，Linux内核版本为3.11.10。每个实验都与几个通用文件系统比较，包括btrfs，ext4，XFS和zfs。误差条和±范围表示95％置信区间。

## 7.1 Microwrites

我们使用元数据和数据密集型微基准评估了微晶体的性能。为了实现文件创建，我们使用TokuBench基准测试[15]在一个扇出为128的平衡目录树中创建了500,000个200字节的文件。结果如图7.1所示.TokuBench也测量了文件系统作为线程的可伸缩性被添加;由于我们的机器具有4个内核，因此我们测量了多达4个线程.BetrFS显示出比其他文件系统高得多的吞吐量。最接近的竞争对手是1线的zfs;随着更多的线程被添加，差距显着扩大。与ext4，XFS和btrfs相比，BetrFS的吞吐量要高出一个数量级。

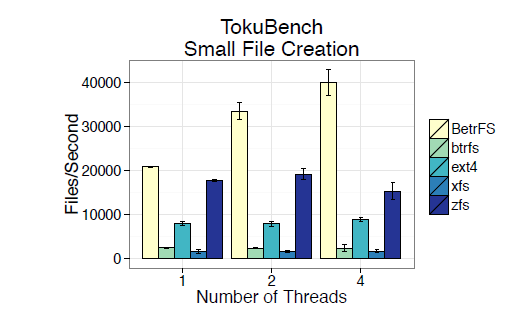


图7.1 使用1,2和4线程创建500,000个200字节文件的总时间。

这种性能差异归因于较少的总写入次数和较少的写入每个字节的查找次数，即较好的小写入聚合。基于blktrace的分析，一个主要区别是写入的总字节数：BetrFS写入磁盘的总MB数量减少了4-10倍，写入请求总数减少了一个数量级。 在其他文件系统中，ext4，XFS和zfs写入的数据量大致相同，但实现了基本写入吞吐量的广泛变化。唯一具有可比写入吞吐量的文件系统是zfs，但是它使用12.7倍的磁盘请求写入了两倍的数据。

为了测量微文件到文件，我们编写了一个自定义的基准测试，在1GiB文件中执行1,000个随机4字节的写入，然后是fsync（）。图7.2列出了结果。 BetrFS比其他文件系统快了两个数量级。

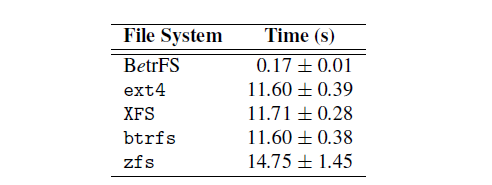


图7.2 以秒为单位的时间在1GiB文件内执行1000个4字节的微字符

这些结果表明，与当前的通用文件系统相比，BetrFS将微晶体的性能提高了一到两个数量级。

## 7.2 大型读写

我们测量了一次读取和写入1GiB文件的吞吐量，一次10个块。我们使用随机数据创建文件，以避免在BetrFS中进行不公正的压缩。在这个实验中，BetrFS受益于压缩密钥，而不是数据。我们注意到，使用压缩和适度压缩的数据，BetrFS可以轻松地超过磁盘带宽。结果如图7.3所示。

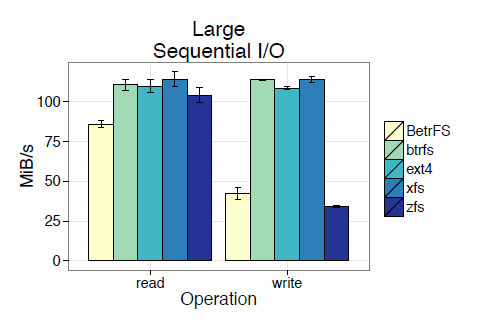


图7.3 文件I/O。

一般来说，大多数通用文件系统可以读取和写入磁盘带宽。在大量连续读取的情况下，BetrFS可以以大约85 MiB/s的速率读取数据。这个读取速率与整个磁盘利用率相当，我们认为这是由于比其他文件系统更不积极的预读。我们相信这可以通过重新调整TokuDB块缓存预取行为来解决。

在大写入的情况下，当前的BetrFS原型实现了低于磁盘吞吐量的一半。原因是每个块写入必须渗透到内部树缓冲区中; 更有效的启发式检测将会检测到大量的流式写入，并直接写入叶节点。作为一个实验，我们手动强制写入空树中的叶子，发现写吞吐量与其他文件系统相当。也就是说，应用这种优化是有点棘手的，因为有许多边缘情况下叶子必须被读取和重写或消息必须被刷新。 我们将这个问题留作未来工作。

## 7.3 目录操作

在本节中，我们将测量BetrFS设计对大型目录操作的影响。图7.4报告了在Linux 3.11.10源代码树上从冷藏缓存开始运行find，grep-r，mv和rm-r所用的时间。grep测试递归搜索字符串“cpu为64”的文件内容“，并且find测试搜索名为”wait.c“的文件。重命名测试重命名整个内核源代码树，删除测试执行源的递归移除。

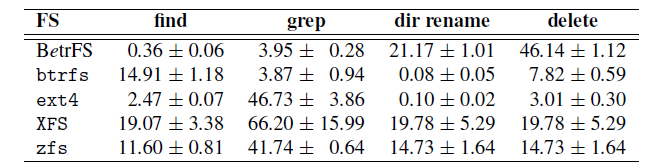


图7.4 目录操作基准测量，以秒为单位。

find和grep基准测试都证明按照全路径按字典顺序排列文件及其元数据的价值，以便相关文件在磁盘上彼此靠近存储。BetrFS可以搜索目录元数据和文件数据比其他文件系统快一个或两个数量级，除了btrfs上的grep，这是可比的。

重命名和删除测试都显示了BetrFS的最坏情况。由于BetrFS不包括从路径名到数据的间接层，因此重命名需要将所有数据和元数据复制到树中的新点。我们还测量了大文件重命名，同样看到了大量的开销-文件中块的数量。虽然已知这个问题的解决方案，比如通过增加一个间接层，我们计划调查技术，可以保留有吸引力的字典位置而不牺牲重命名和删除性能。

## 7.4系统调用Nanobenchmarks

最后，图7.5显示了测量各种系统调用时间的nanobenchmark的时间。由于这个nanobenchmark是温暖的缓存，它主要练习VFS层。BetrFS接近成为开放，阅读和统计的最快的文件系统。在chmod，mkdir和unlink之间，BetrFS在这个包的中间。

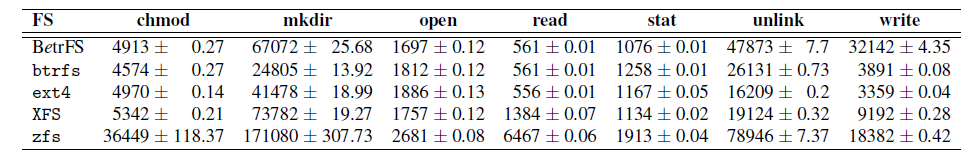


图7.5 执行一系列通用文件系统调用的平均周期时间

我们目前的写入系统调用的实现在这个基准测试中似乎很慢，因为正如5.1节所述，即使正在写入的页面在缓存中，也会在BetrFS问题中写入数据库的upsert。这对于页面 不是经常写的，但是在这个基准测试中并不是这样。

## 7.5 空间开销

BetrFS中的分形树索引实现包含一个缓存表，缓存树节点。Cachetable内存是有界的。当内存超过低水位时，BetrFS会触发背景刷新，并在高水位时强制回写。高位水印当前设置为总系统内存的八分之一。这是可配置的，但是我们发现额外的缓存内存对我们的工作负载没有什么性能影响。

没有单个规则管理BetrFS磁盘使用情况，因为在删除，重命名和覆盖操作之后，陈旧的数据可能会保留在非叶节点中。后台清理器线程尝试每秒从5个内部节点中刷新未决数据。这造成了BetrFS磁盘使用量的波动，但是开销在休息时迅速下降。

为了评估BetrFS磁盘的占用空间，我们写了几个大的不可压缩的文件，删除了这些文件的一半，然后启动了一个*Bɛ-tree*的刷新。每次操作之后，我们使用基础ext4分区上的df来计算BetrFS磁盘的使用情况。

如图7.6所示，向BetrFS写入新数据的开销非常小。然而，对于删除，BetrFS会为每个文件块发出一个upsert，这对于BetrFS的占用空间几乎没有影响，因为过时的数据被延迟地回收。 刷新之后，不管数据量如何，磁盘开销都不到3GiB。

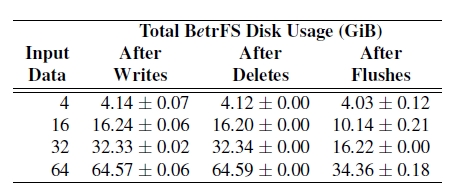


图7.6 在GiB中测量的BetrFS磁盘使用情况

# 8 结论

BetrFS原型演示了写入优化索引对于文件系统开发人员来说是一个强大的工具。在某些情况下，BetrFS将传统设计的性能提高了一个数量级，推动了先前技术的发展。尽管如此，在某些情况下还需要额外的工作，例如对大型流式I/O进行进一步的数据结构优化以及对目录进行高效的重命名。我们的研究结果表明，进一步的整合和优化工作可能会产生更好的性能结果。

参考文献

[1] ANDERSEN, D. G., FRANKLIN, J., KAMINSKY,M., PHANISHAYEE, A., TAN, L., AND VASUDEVAN, V. Fawn: A fast array of wimpy nodes. In Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (2009), pp. 1–14.

[2] APACHE. Accumulo.http://accumulo.apache.org.

[3] APACHE. HBase. http://hbase.apache.org.

[4] BENDER, M. A., FARACH-COLTON, M.,FINEMAN, J. T., FOGEL, Y. R., KUSZMAUL,B. C., AND NELSON, J. Cache-obliviousstreaming B-trees. In Proceedings of the 19th Annual ACM Symposium on Parallel Algorithmsand Architectures (SPAA) (2007), pp. 81–92.

[5] BENT, J., GIBSON, G. A., GRIDER, G.,MCCLELLAND, B., NOWOCZYNSKI, P., NUNEZ,J., POLTE, M., AND WINGATE, M. PLFS: A checkpoint filesystem for parallel applications. In Proceedings of the ACM/IEEE Conference on High Performance Computing(SC) (2009).

[6] BLOOM, B. H. Space/time trade-offs in hash coding with allowable errors. Commun. ACM 13, 7 (1970), 422–426.

[7] BONWICK, J. ZFS: the last word in file systems.https://blogs.oracle.com/video/entry/ zfs\_the\_last\_word\_in, Sept. 14 2004.

[8] BRODAL, G. S., DEMAINE, E. D., FINEMAN,J. T., IACONO, J., LANGERMAN, S., AND MUNRO, J. I. Cache-oblivious dynamic dictionaries with update/query tradeoffs. In Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms (SODA) (2010), pp. 1448–1456.

[9] BRODAL, G. S., AND FAGERBERG, R. Lower bounds for external memory dictionaries. In Proceedings of the 14th Annual ACM-SIAM symposium on Discrete Algorithms (ACM) (2003), pp. 546–554.

[10] BUCHSBAUM, A. L., GOLDWASSER, M.,VENKATASUBRAMANIAN, S., AND WESTBROOK, J. R. On external memory graph traversal. In Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA) (2000), pp. 859–860.

[11] CARD, R., TS’O, T., AND TWEEDIE, S. Design and implementation of the Second Extended Filesystem. In Proceedings of the First Dutch International Symposium on Linux (1994), pp. 1–6.

[12] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH,W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E.Bigtable: A distributed storage system for structured data. ACM Transactions on Computer Systems (TOCS) 26, 2 (2008), 4.

[13] CHAZELLE, B., AND GUIBAS, L. J. Fractional cascading: I. A data structuring technique. Algorithmica 1, 1-4 (1986), 133–162.

[14] DOUTHITT, D. Instant 10-20% boost in disk performance: the “noatime” option. http://administratosphere.wordpress. com/2011/07/29/instant-10-20-boost-indisk-performance-the-noatime-option/,July 2011.

[15] ESMET, J., BENDER, M. A., FARACH-COLTON,M., AND KUSZMAUL, B. C. The TokuFS streaming file system. In Proceedings of the 4th USENIX Workshop on Hot Topics in Storage (HotStorage) (June 2012).

[16] File system in userspace.http://fuse.sourceforge.net/.

[17] GOOGLE, INC. LevelDB: A fast and lightweight key/value database library by Google. http://code.google.com/p/leveldb/.

[18] GRAY, J., AND REUTER, A. Transaction Processing: Concepts and Techniques. Morgan Kaufmann, 1993.

[19] HITZ, D., LAU, J., AND MALCOLM, M. File system design for an NFS file server appliance. Tech. rep., NetApp, 1994.

[20] LAKSHMAN, A., AND MALIK, P. Cassandra: a decentralized structured storage system. ACM SIGOPS Operating Systems Review 44, 2 (2010),35–40.

[21] LEE, C., SIM, D., HWANG, J., AND CHO, S.F2FS: A new file system for flash storage. In Proceedings of the USENIX Conference on File and Storage Technologies (FAST) (2015).

[22] LIM, H., FAN, B., ANDERSEN, D. G., AND KAMINSKY, M. Silt: A memory-efficient,high performance key-value store. In Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (2011), pp. 1–13.

[23] O’NEIL, P., CHENG, E., GAWLIC, D., AND O’NEIL, E. The log-structured merge-tree(LSM-tree). Acta Informatica 33, 4 (1996),351–385.

[24] QUICKLZ. Fast compression library for c, c#, and java. http://www.quicklz.com/.

[25] REN, K., AND GIBSON, G. A. TABLEFS:Enhancing metadata efficiency in the local file system. In USENIX Annual Technical Conference(2013), pp. 145–156.

[26] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. ACM Transactions on Computer Systems 10, 1 (Feb. 1992), 26–52.

[27] SEARS, R., CALLAGHAN, M., AND BREWER,E. A. Rose: compressed, log-structured replication. Proceedings of the VLDB Endowment 1, 1 (2008), 526–537.

[28] SEARS, R., AND RAMAKRISHNAN, R. bLSM: a general purpose log structured merge tree. In Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (2012), ACM, pp. 217–228.

[29] SELTZER, M., BOSTIC, K., MCKUSICK, M. K.,AND STAELIN, C. An implementation of a log-structured file system for UNIX. In Proceedings of the USENIX Winter 1993 Conference Proceedings (San Diego, CA, Jan.1993), p. 3.

[30] SELTZER, M., SMITH, K. A., BALAKRISHNAN,H., CHANG, J., MCMAINS, S., AND PADMANABHAN, V. File system logging versus clustering: A performance comparison. In Proceedings of the USENIX 1995 Technical Conference Proceedings (New Orleans, LA, Jan. 1995), p. 21.

[31] SHETTY, P., SPILLANE, R. P., MALPANI, R.,ANDREWS, B., SEYSTER, J., AND ZADOK, E. Building workload-independent storage with VT-trees. In FAST (2013), pp. 17–30.

[32] SWEENY, A., DOUCETTE, D., HU, W.,ANDERSON, C., NISHIMOTO, M., AND PECK, G. Scalability in the XFS file system. In Proceedings of the 1996 USENIX Technical Conference (San Diego, CA, Jan. 1996), pp. 1–14.

[33] TOKUTEK, INC. TokuDB: MySQL Performance,MariaDB Performance. http://www.tokutek. com/products/tokudb-for-mysql/.

[34] TOKUTEK, INC. TokuMX—MongoDB Performance Engine. http://www.tokutek. com/products/tokumx-for-mongodb/.

[35] WANG, P., SUN, G., JIANG, S., OUYANG, J.,LIN, S., ZHANG, C., AND CONG, J. An efficient design and implementation of LSM-tree basedkey-value store on open-channel SSD. In Proceedings of the Ninth European Conference on Computer Systems (2014), EuroSys ’14, pp. 16:1–16:14.