日志文件系统的设计与实现

71114210 杜臻

毕业设计论文翻译

2018年春季

日志文件系统的设计与实现

杜臻毕设论文翻译

原作者

Electrical Engineering and Computer Sciences, Computer Science Division
University of California
Berkeley, CA 94720

mendel@sprite.berkeley.edu, ouster@sprite.berkeley.edu

概要

这篇文章描述了一种磁盘管理的新技术，我们称之它为日志文件系统。一个日志文件系统可以用类似于日志的结构向磁盘中序列化地写入所有修改，因此来提升数据写入的速度和磁盘崩溃之后的恢复速度。所谓日志是磁盘上的唯一一种结构；它包含索引信息，以便可以有效地从日志中读回文件。为了可以充分管理磁盘上的大量空余空间来加快写入过程，我们将日志分割成一个个段（segments）并且使用段清除器（segment cleaner）来压缩高度碎片化的现有数据。我们使用了一系列模型来证明这种既有开销又有收益的碎片段清除策略的高效性。我们已经实现了日志文件系统的原型，我们把这个日志文件系统称作Sprite LFS；它比现有的Unix文件系统在小文件写入方面的性能提高了一个数量级，并且在大量写入和读取的性能上也是完全不低于现有Unix文件系统。即便在“段碎片清除”上需要额外的开销，Sprite LFS在文件写入上也可以发挥出70%的磁盘带宽，而现有的Unix file system只能发挥5-10%的带宽。

1、介绍

在过去的10年中，CPU的速度已经飞快地提升，但是磁盘的访问速度却提升得很慢。这个趋势会在很长的一段时间持续，这会使得越来越多应用程序的性能卡在磁盘上。为了减少这个磁盘性能的不利影响，我们提出了一个新的磁盘存储管理系统，所谓日志文件系统。这个文件系统可以极大提升磁盘利用率。

日志文件系统建立在一个假设之上，那就是文件都是尽可能缓存在主存之中的，而随着主存容量越来越大，使用缓存来处理读请求会越来越高效。所以磁盘IO会更加依赖于磁盘写入的质量。日志文件系统通过一种叫做日志的序列化（顺序）结构来向磁盘中写入新的信息。这种方法通过消除几乎所有的查找（seek）过程来显著提高写入性能。日志的序列化（顺序）特性还允许更快的崩溃恢复：当前的Unix文件系统通常必须在崩溃后扫描整个磁盘以恢复一致性，但是日志结构化的文件系统只需要检查日志的最新部分就可以得到一样的效果。

日志的概念并不是新出现的，现有的文件系统已经通过引入日志作为一种辅助手段来提升写入效率与恢复速度。但是这些文件系统仅仅使用日志来进行临时存储；信息的持久化存储还是在磁盘上的、传统的、随机访问的结构中。相反，日志文件系统将日志中的信息持久化地存储。日志中包含了文件的索引信息，文件可以获得更高的读回效率。

为了使得一个文件系统可以高效运作，必须保证有大量的（整块）空余空间可以写入新数据，这是日志文件系统最大的挑战。此篇文章，我们通过采用不断压缩整理现有数据的方式来生成空段，以此来达到整理大段空余空间的目的。为此我们引入了段（segment）的概念，而段清理（segment cleaner）进程通过不断整合压缩现有数据来生成新的段（译者注：我认为这里应该就是类似于“磁盘碎片整理”的过程）。我们模拟了不同的整理策略并且在权衡利弊之后发现了一个简单但是有效的算法：将老的、更新更慢的数据和快速变化的数据分开处理。

我们已经构造了一个日志型文件系统的雏形，叫做Sprite LFS，现在已经作为Sprite网络操作系统（Sprite network operating system）的一部分。测试程序已经表明，Sprite LFS在小文件的写入上遥遥领先于Unix。甚至在其他的使用场景，比如说读和大文件的访问，Sprite LFS在除了一个场景下（在随机写之后顺序读取文件），都有不弱于Unix的表现。并且我们测试了在实际应用的生产环境中长时间的段清理（cleaning）开销。总之，Sprite LFS在写入新数据上可以利用磁盘65-75%的性能（剩下的用来段清理（cleaning））。相比之下，Unix系统只能利用5-10%的磁盘带宽，剩下的时间都是在做数据的搜寻（seeking）。

在这个文章的剩下部分可以组织成6个部分。第二章将回顾上世纪90年代的文件系统设计。第三章讨论了日志文件系统的设计和SpriteLFS的架构，并特别关注于段清理（cleaning）的机制。第四章将描述Sprite LFS的崩溃恢复系统。第五章将会使用测试程序来评估Sprite LFS的性能，以及对于段清理开销的长时间测量。第六章将SpriteLFS和其他的文件系统相比较，第七章是一个总结。

2、上世纪90年代的文件系统设计

文件系统的设计主要有两个推动力：技术，它提供了一系列文件系统所需要的基本构建模块；以及使用场景，他决定了哪些操作需要被有效率地执行。这个模块总结了已经在发生的技术变化并且说明它们在文件系统设计上所能带来的影响。并且这个章节将会将会讲述会影响Sprite LFS设计的使用场景并且展示当今的文件系统是在使用场景和技术大环境的变化下处理得并不得当。

2.1、技术

在文件系统的设计中，有三个技术对其影响很大：处理器、硬盘、主存。处理器很重要是因为它们的速度正原来越快地增长，尤其在上世纪90年代之后。这给予计算机系统的其他部分很大的压力，它们也必须得到相同的提升，否则将成为系统的短板，阻碍计算机系统的发展。

硬盘技术也在以很快的速度提升，但是主要的提升体现在价格和容量而不是性能上面。硬盘的性能分为两个部分：传输带宽和访问时间（译者注：我觉得这类似于硬盘的寻道时间）。虽然这两个部分都在不断提升，但是提升的速度都远低于CPU的提升速度。磁盘传输带宽可以通过使用磁盘阵列和并行磁盘来大幅提高，但访问时间似乎没有大的改进（这是由很难改进的机械运动决定的）。这就意味着如果一个程序导致了一系列小的磁盘IO，中间穿擦着数据的寻找（译者注：这是一个有需要大量需要时间的使用场景），那么这个应用程序在未来的10年都不会有什么性能提升，即便有更快的处理器。

第三个技术是主存，主存的容量再以越来越快的速度增长。现代的文件系统都会将最近使用的文件数据缓存在主存中，并且越大的主存就意味着越大的文件缓存。这对文件系统的操作造成了两个影响。首先更大的文件缓存接管了大量的读请求来降低磁盘问负载。而大多数的写操作都要真正落实到磁盘上来保证可靠性，所以磁盘的性能将越来越由写操作决定。

大文件缓存的第二个影响是他们可以作为写缓存区，在这个缓存中大量的已修改的文件块可以在先暂时收集在这里。缓冲区让更高效的块写入成为可能，比如说我们可以调整写入磁盘的顺序，使得写入磁盘只需要一次寻道（seek）就可以解决。当然，写缓冲区也有问题，那就是当系统出现崩溃的时候大量的数据丢失。在这篇文章中我们假设崩溃是偶然的，在每次崩溃中利用几秒钟或几分钟的工作去恢复是可以接受的。对于需要更好崩溃恢复的应用程序，可以使用非易失性RAM作为写缓冲区。

2.2、工作场景

对于计算机应用程序来说，有几个不同的文件系统使用场景是很常见的。一种最难的工作场景是办公室和工程环境。办公室和工程应用以小文件访问为主；很多的研究已经表明，在这种场景下文件的场景通常只有几kb。小文件通常会导致小的磁盘随机IO，在这些小文件的删除和和创建上又有系统的元数据更新操作（一种数据结构来定位文件的属性和块位置）。

一些以大文件顺序访问为主的使用场景，比如说大型超级计算机的使用场景，也提出了一些有意思的问题，但是这个不是文件系统该 解决的问题。

有一系列的技术已经出现，来保证文件顺序地放在磁盘上以供读取，因此I/O性能往往受I/O和内存子系统带宽的限制，而不受文件分配策略的限制。在日志文件系统的设计中，我们将主要的经历放在提升小文件的访问效率上，而让硬件设计者来提升大文件的访问带宽。幸运的是，在Sprite LFS使用的技术在大文件和小文件中都有很好的表现。

2.3、现有的文件系统的问题

现有的文件系统受困于两个主要的问题，使得它们在处理上世纪90年代的一些技术和使用场景的时候有明显的困难。首先他们将数据广泛地分散在磁盘中，这会导致非常多细碎的访问。例如，Berkeley Unix快速文件系统（Unix FFS）在将每个文件按顺序排列在磁盘上是相当有效的，但它在物理上将不同的文件分开。此外，文件的属性（“inode”）与文件的内容是分开的，就像包含一个文件名的目录的入口一样。在Unix FFS中创建一个文件需要5次I/O，其中每次I/O之前都需要一次寻道（seek）操作。这5次IO分别包括两次对文件属性的不同访问，以及分别对文件数据、目录数据、目录属性的访问。在这样的文件系统中写入一个小文件，只有不到5%的磁盘潜在带宽用来处理新数据；剩下的时间都是花费在寻道（seeking）上。

第二个问题就是现有的文件系统都趋向于同步地写入：应用程序都必须等待写入过程全部完成，而不是把写入过程放到后台，继续处理其他的事情。举一个例子，即便Unix FFS的每一个文件数据块是异步写入的，但是文件系统的元数据信息，比如目录和inode，写入是同步的。在很多小文件的使用场景中，磁盘传输性能主要由同步的元数据写入决定。同步写入和应用程序性能是密切相关的，如果写入拖慢了应用程序的性能，那么应用程序就不会受益于更快的CPU。并且这种同步写入的设定也让写缓冲区不能被充分利用。不幸的是，网络文件系统，比如NFS，也引入了一个之前不存在的额外同步操作。这是的灾备恢复过程被简化，但是降低了写入性能。在这篇文章中，我们将Unix FFS作为现代文件系统设计的典范，并且让其与日志文件系统做对比。Unix FFS的设计在这里被使用，是因为它有非常

完整的文档，使用在非常流行的Unix操作系统中。而在这个章节中提到的很多问题也并不仅限于Unix FFS，在其他的文件系统中都可以找到。

3、日志文件系统

日志文件系统设计的初衷就是去提升写性能，它在文件缓存中存储一系列的文件系统修改，然后将所有的修改通过一次写操作顺序写入磁盘中。写入磁盘中的信息包括文件的数据块、属性、索引块、目录以及其他用来文件系统管理的信息。在包含了很多小文件的使用场景中，一个日志文件系统可以将很多小的同步随机写入转变成传统文件系统的异步顺序传输。这样子就可以利用100%的磁盘带宽。

虽然日志文件系统的基本想法非常容易，但是为了实现日志记录方法的潜在优点，必须解决两个关键问题。第一个问题是如何在log中取出信息；这是章节3.1的主题。第二个问题是怎么管理磁盘的空余空间，这样子大量的空余空间就可以整理出来用以写入新数据。这是一个更加难的问题；在3.2-3.6中进行讲述。表格1包含了所有在磁盘上的、Sprite LFS用以解决上述困难的数据结构的总结；在后续的章节中，这些数据结构会被集中讨论。

数据结构	目的	位置
Inode	定位文件块的位置，设定保护位，修改时间，等等	日志
Inode Map	定位inode在log中的位置，保存最后访问的时间和版本号	日志
Indirect Block	定位大型文件的block	日志
Segment summary	段的元数据（段属于的文件号和在文件中的偏移）	日志
Segment usage table	记录段中已经存在的数据大小，保留段的最后一次写入时间	日志
Superblock	一些静态配置，包括段的数量和段的大小	固定
Checkpoint region	定位inode map和segment usage table的块的位置，记录日志的最后检查点	固定
Directory change log	记录目录操作，以保持inode引用计数的一致性	日志

表格1：Sprite LFS中重要数据结构总结

3.1、文件位置与读取

虽然“日志文件系统”通常建议顺序扫描日志来从日志中获取数据，但是这不是Sprite LFS中所使用的方法。我们的目标是在写入性能上达到或者超

过Unix FFS的水平。为了完成这个目标，Sprite LFS在日志中输出索引结构以允许随机访问检索。这种在Sprite LFS中使用的基本结构与Unix FFS是相同的：对于每一个文件都有对应一个叫做inode的数据结构，这个结构包括属性以及文件的前10个块在磁盘中的位置；对于大于10个块的文件来说，inode中也有一个或者多个Indirect Block在文件中的位置，这些Indirect Block又包含了更多数据块的在磁盘中的位置和其他Indirect Block在磁盘中的位置。只要一个文件的inode被找到，那么在读取一个文件需要的IO次数上，Sprite LFS和Unix FFS是一样的。在Unix FFS中每个inode位于磁盘上的一个固定位置，给定一个文件的标识号（译者注：这可能就是类似于Linux fid的东西），一个简单的计算就会得到该文件的inode磁盘地址。相反，Sprite LFS并不把inode放在固定的位置；他们被放在日志中。Sprite LFS使用一个叫做inode map的数据结构来维护每一个inode的最近存储位置。通过file的标识号（fid），inode map可以映射到inode在磁盘中的存储位置。Inode map被分成一个个数据块，然后写入日志中，一个存储位置固定的checkpoint region（在前面的表格中有提到）会记录所有inode map的数据块在磁盘中的位置。幸运的是，inode map通常来讲都非常小巧，以至于可以把活跃的映射关系缓存在内存中： 这样子就不需要太多的磁盘访问了。

图1表现了分别在Sprite LFS与Unix FFS存储了两个新文件在两个不同目录的磁盘布局。虽然这两种布局具有相同的逻辑结构，但是日志结构文件系统的布局更为紧凑。因此，Sprite LFS的写入性能比Unix FFS好得多，而其读取性能也一样好。

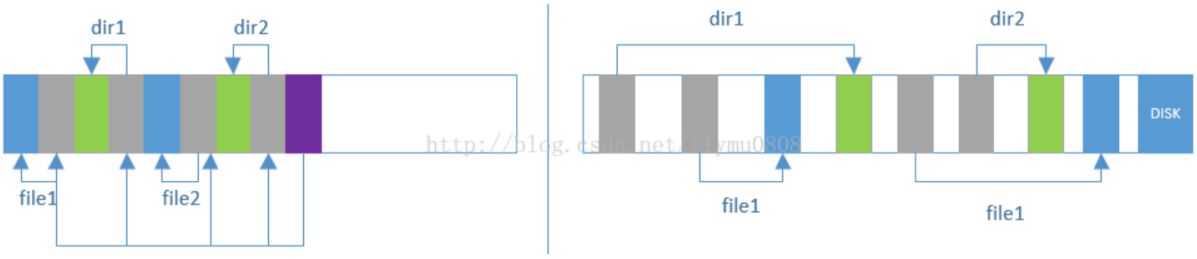

图1—Sprite LFS与Unix FFS之间的比较

（因为原图有比较严重的印刷错误，所以，这里采用了来自于互联网的图片。其中紫色的是inode map，灰色的是inode，绿色的是目录文件，蓝色的是普通文件）

这个例子展示了当创建两个只有一个数据块大小的文件dir1 / file1和dir2 / file2时，被修改的磁盘块的样子。每个系统必须为file1和file2写入新的数据块和inode，以及包含的目录的新数据块和目录的inode（目录也当做一种文件）。Unix FFS需要10次非顺序写入新信息（新文件的索引节点每次写入两次以便缓解崩溃恢复），而Sprite LFS在一次大写操作中执行操作（译者注：因为是一次顺序写入，所以存储非常紧凑）。读取两个系统中的文件需要相同数量的磁盘访问。 Sprite LFS还写出新的inode map块来记录新的inode位置。

3.2、空闲空间管理：段

日志文件系统最大的设计上的困难就是空闲空间的管理。这种管理的目标是要维护大的空闲空间来写入新的数据。在一开始，所有的可用空间在磁盘上就是一个整块，当日志已经写到磁盘的末尾的时候，这个时候的可用空间就会是一个个小碎片分布在磁盘的各个角落，这种小碎片是由文件的删除和重写（对文件进行修改会改变文件的大小）引起的。

对于这个问题，文件系统有两种解决方式，threading（穿插）和copying（复制）。所谓threading，就是将已经存在的数据放在原地，新加入的日志先优先放到碎片中，然后使用一个指针将不同碎片中的相同日志连接在一起。这样子做会使得整个文件系统的碎片化变得越来越严重，并且大文件的写入将会出现很大的问题。在这种方式下，日志文件系统将不会比传统的文件系统更快。第二种方式就是将数据拷贝出来之后重新紧凑地存储。这样子就可以空余出大量的整块空间来进行数据的写入了。这篇文章我们将假设已有数据已经重新紧凑地写入在日志的头部，也可以将其移动到另一位日志文件系统来来形成日志层次结构，或者将其移动到完全不同的文件系统或者归档中。copying的缺点就是开销，特别是长期存在的文件；

有这么一个最简单的案例：日志在磁盘上面循环工作，现有的数据会一次次拷贝回日志，来做紧凑的存储整理，所有的长时间的存在的文件将必须被一次次在日志中一次次被复制。

Sprite LFS使用threading和copying两种方式的组合。磁盘被分成一个个固定大小的空间，被称作段（segment）。每一个给定的段都是从头到尾顺序写的，在段可以被重写之前，所有已经存在的数据必须被复制出去。然而，日志被一段一段地拆分和连接；如果系统可以收集到长时间存在的数据并且整合到一系列段中，那么这些段中的数据将会在整理的时候被跳过防止一次次重复地复制整理。段的大小将会设置地足够大，这样子对于段位置的寻找在整个数据传输中时间上是可以忽略不计的。这使得整个段的操作几乎可以利用磁盘的全部带宽，而不管段的访问顺序如何。 Sprite LFS当前使用512千字节或1兆字节的段大小。

3.3、段整理策略

将现有的数据复制到段之外的过程叫做段整理（segment cleaning）。在Sprite LFS中这一个简单包含了三个步骤的过程：将一系列的段读入内存，识别出里面已经存在的数据，将已经存在的数据写会更少的空白段中。在这个工作做完之后，之前写入内存的那些段都可以认为是已经清理干净的，可以用来写入新的数据，并且可以进行新的一轮整理操作。

作为段清理的一部分，必须能够识别一个段中的哪些块是有数据的，这样子才可能进行整理的操作。除此之外还必须识别出，每个块所属的文件和这个块在文件中的位置；这些信息必须是非常必要的，因为这样子才可能更新inode来重新定位文件对应的块所在的位置。Sprite LFS通过在每个段中加入段摘要块来解决这两个问题。摘要块会标记写入段的每条信息；例如，对于每个文件数据块，摘要块都包含该块的文件编号（fid）和块编号（这个块在文件中的位置）。当需要多个日志写入来填充段时，段可以包含多个段摘要块。 段摘要块在写入过程中几乎没有额外开销，但是它在灾备恢复和段整理的期间很管用。

Sprite LFS也使用段摘要信息来正将发挥作用的、已经写了数据（live）的数据块从已经被删除和覆盖的数据块中分离出来。所以说，对于一个块来说，我们可以通过查询文件的inode和间接块（之前提到的indirect block）来看这个块是不是还在被文件引用，从而来看这个文件是不是还在发挥作用。Sprite LFS通过在inode map中为了每个文件维护一个版本号来优化上

述块的检查过程；当文件经历任何操作，进行删除或者truncate到长度0的时候，这个版本号都会增加。版本号与inode编号组成一个文件内容的唯一标识符（uid）。段摘要信息块会记录在段中每一个文件块（block）的uid；如果这个文件块的uid在inode map中没有找到，那么这个块我们就可以永远忽略掉了（当做空的，不进行合并整理）。

我们在这个过程中可以看出，在Sprite中并没有空块表（free-block list）或者位图（bitmap）来记录空闲空间。除了可以节省内存和磁盘空间之外，这种对于上述两种方法的抛弃也简化了崩溃恢复。如果存在这些数据结构，则需要额外的代码来记录对结构的更改并在崩溃后恢复一致性。

3.4、段整理策略

对于上文所述的这种机制，有四个策略问题需要去解决：

（1）什么时候段整理策略可以被执行？有一种选择是我们可以让其不间断地在后台运行，并且给予一个比较低的优先级，还有一种选择是我们在晚上进行，还有一种选择是让磁盘空间即将枯竭的时候再进行。

（2）一次我们要整理多少个段？段整理是提供了一个重新组织磁盘上文件的机会；一次整理越多的段，那就有更多的机会进行组织调整。

（3）哪一个段需要被整理？最明显的选择是碎片化最严重的段要被整理，但是这明显不是最好的选择。

（4）对于被合并的数据块，要如何重新组合成段并写回磁盘？一种可能的方式是让将来可能会一起读取的文件放到一个段中，比如将同一个目录下的文件放到一个输出段中。还有一种可能性是让段按照最后一次修改时间排序，然后然访问最后一次修改时间相似的块放在一个段中，我们把这种方式叫做年龄排序（age sort）。

至今我们还没有有条不紊地解决上诉策略的前两个问题。当空白段的数量小于一个阈值的时候（通常为十几个段时）Sprite LFS开始段整理的工作。它一次清理几十个段，直到清理出来的段超过一个阈值（通常50-100）。

Sprite LFS的整体性能对阈值的确切选择似乎并不敏感。相比之下，第三和第四个策略是至关重要的：根据我们的经验，它们是决定日志文件系统

性能的主要因素。此章节的其余部分将讨论我们分析哪些段需要整理以及如何对整理好的数据块进行分组。

我们使用一个叫做写开销（write cost）的术语来比较段整理策略。写入成本是写入新数据的每个字节的磁盘占用的平均时间量，包括所有段整理开销。如果没有段整理开销，写入成本可以表示为所需时间的倍数，并且数据可以在没有寻道时间或旋转延迟的情况下以其全带宽写入。比如说，写入成本倍数是1.0的话就代表非常好：这代表了数据可以利用磁盘的全部带宽写入数据。如果写入开销是10，那就意味着只有十分之一的带宽用来真正写入数据；剩下的磁盘时间都用来数据搜索（寻道），旋转延迟以及段整理上。

如果一个日志文件系统的段比较大，那么搜索（seeks）以及旋转延迟就在这个写入和段整理的过程中忽略不计，所以说写开销就是磁盘读和写的所有字节除以新数据的字节数。这个开销是由要被整理的段的利用率决定的（一个段中的碎片越多，那么这个段的利用率就越低）。在一个稳定的情况下，我们要为每一个段的新数据整理出一个一个新的段来容纳它。为了完成这个工作，我们要全部读取N个段，然后写回N*u个段的整理好的数据（u<1，u小代表整理之后的数据越小，u在文章中被称作段利用率，u越小，write cost就越小，也就意味着段整理对于磁盘的写入占用越小）。下面这段公式就是u与写开销之间关系的证明：

$$\text{write cost} = \frac{\text{total bytes read and written}}{\text{new data written}}$$

$$= \frac{\text{read segs} + \text{write live} + \text{write new}}{\text{new data written}}$$

$$= \frac{N + N*u + N*(1-u)}{N*(1-u)} = \frac{2}{1-u} \qquad (1)$$

上面的计算我们使用的是一个比较保守的假设，那就是一个段必须被完整读出来才能进行段整理的工作，但是实际上这个过程可以加快很多，我们可以只读取一个段中有用的（live）的数据块（有些数据块因为文件删除

等操作已经没有用了），特别是如果利用率非常低（我们还没有在Sprite LFS中尝试过）。如果一段整理完之后没有产生任何有用的数据块，也就是直接产生空白数据段（u=0）并且他在之后完全没有被读取，那么写开销就是1.0。

图三展示了写开销和u之间的函数关系。值得一提的是，Unix FFS在小文件的场景下可以利用5-10%的磁盘带宽，写开销大概是10-20。在日志和延迟写的加持下，写开销可以缩小到4，可以利用25%的磁盘性能。在图三中我们可以看到，我们必须让 u（利用率）小于0.8才能让日志文件系统超过Unix FFS；u必须小于0.5才可以超过提升之后的Unix FFS。

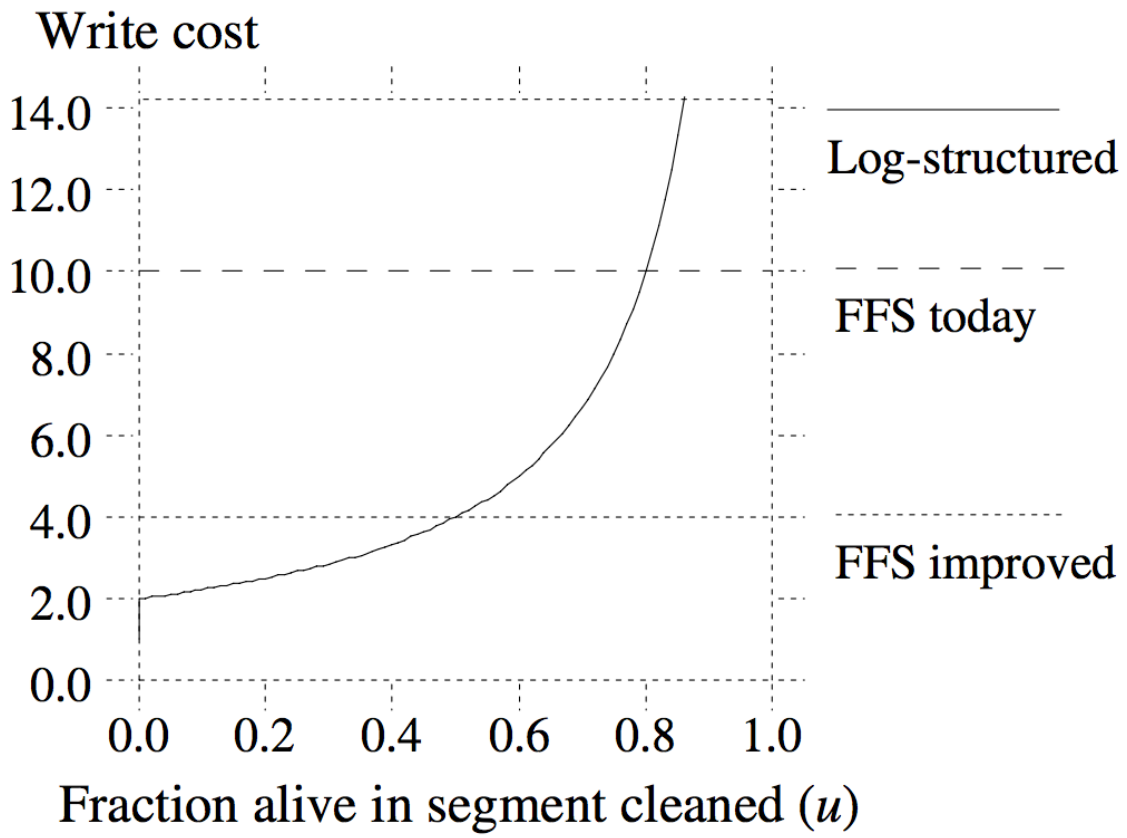

图3—在小文件上的写开销与u（段利用率）之间的关系（右侧几条线是图例）

值的一提的是这里提到的u（段利用率）并不反应磁盘中有用（live）数据的总体情况的一个分数，他只反映了已经被清理了的段中有用（live）数据的一个分数。对于文件利用的差异将会导致一些段的段利用率比其他的段要低；这些段的利用率会低于磁盘段利用率的平均水平。

但是即便如此，日志文件系统的性能依旧可以通过减少磁盘的总体利用率在得到提升。在磁盘使用量较小的情况下，被整理的段的有用（live）数据也会比较少，会导致较低的写入开销。日志文件系统在开销和性能之间做了一个平衡：如果磁盘空间未充分利用，则可以实现更高的性能，但每个可用字节的成本很高；如果磁盘容量利用率增加，则存储成本降低，但性能也降低。这个在日志和空间利用率之间的权衡并不是日志文件系统独有的。比如，Unix FFS只允许90%的磁盘空间被文件利用。剩下的10%保持空闲来运行空间分配算法可以被高效运行。

3.5、测试模型结果

我们创造了一个简单的文件系统模型，这样子我们就可以在可控制的条件下分析不同的存储策略。模拟器的模型并不反映实际的文件系统使用模式（它的模型比实际要严苛得多），但是它帮助我们理解随机访问模式和局部性的影响，这两种都可以被利用来降低清理成本。我们模拟一个文件系统中有固定数量的4KB的文件，并且设定一个特定产生的磁盘总体利用率。在每一步中，模拟器都会使用下面的两种随机访问模式来像其中一个文件写入新数据：

均匀（Uniform）：每个文件在每一步中都会被等概率地被选择到。

不均匀（Hot-and-cold）：文件被分成两组。一个包含了10%的文件；它们被叫做热文件因为这些文件会在90%的时间被选择。另外一个组叫做冷文件组；它包含了90%的文件，但是他们只会在10%的时间上被选择。在每一个组中，每个文件都会被等概率地选取。这种访问模式模拟了一个简单的局部性模式。

在这个模拟的过程中，磁盘整体的利用率是恒定的并且不包含读操作。整个模拟器会一直运行直到所有的空段都消耗殆尽，然后模拟一系列的段整理操作直到一个超过阈值的空段被整理出来。在每一个模拟中，只有当写入成本稳定了之后模拟器才会运行，并且移除一开始的冷启动过程。

图四在图三的理论数据上展示了两组模拟过程的实际结果。在“LFS uniform”中模拟了均匀访问的情况。段整理器使用了一个简单的策略，它每次都选择利用率最小的段来进行清理。将数据整理并写会硬盘的时候，

段整理器将不会对数据进行重新组织：数据块将会按照他们在正在被清理的段中出现的顺序写回（对于随机等概率的访问来说，对于写回数据的重新组织并不会带来任何提升）。

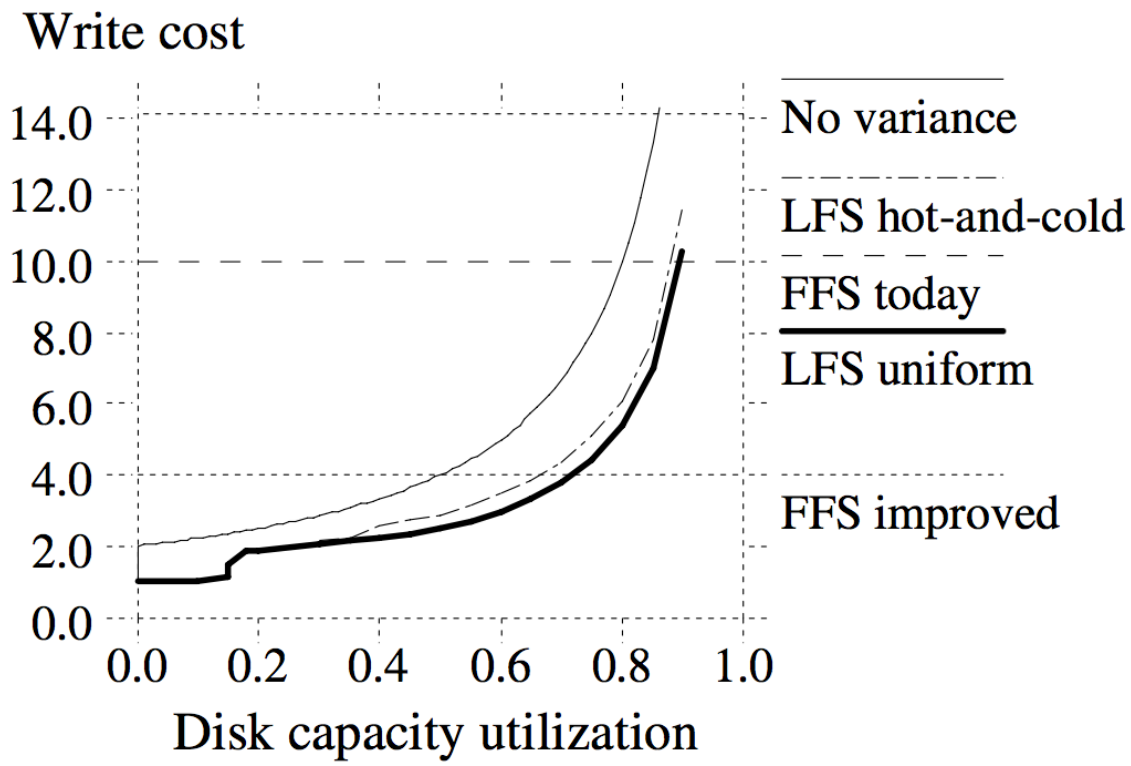

图4—最初的模拟结果

标记为“FFS today”和“FFS improved”的曲线（其实就是一个水平直线）也是从图三中复制下来的，方便比较。被标记为No variance的曲线表明了所有段的段利用率都一样的情况，是理想状态。而另外两个线条分别是日志文件系统在两种不同的随机访问模型下的写开销。

即便采用随机的访问模式，段利用率的差异也会使得写开销相比公式（1）算出的理论利用率更低。比如在75%的磁盘整体利用率下，被整理的段的平均利用率只有55%。当磁盘的整体利用率小于20%的情况下，写开销会降低到2.0之下；这意味着一些清理的段根本没有活动块，因此不需要被读入。

“LFS hot-and-cold”曲线展现了当访问有一定局部性时候的写开销。在这个曲线中的段整理策略基本上与“LFS uniform”一样，除了数据块在被写回磁盘之前会按照最近访问时间的顺序排序。这就意味着冷数据和热数据会在不同的段中；我们认为这种方法会导致段利用率的期望双峰分布。

图四体现出来了一个超过预期的令人惊讶的结果。拥有良好局部性访问的性能竟然低于没有局部性访问的性能！我们不断调整访问的局部性（比如95%的数据被5%概率访问）并发现访问的局部性越高，性能就越差。在图5中展示了这种不符合直觉的结果出现的原因。在较为贪婪的策略下，一个段并不会清理直到它在所有段中的段利用率最低。因此每一个段的利用率都会在最终掉到某一个阈值之下，包括很多很多冷数据段。不幸的是，段利用率在冷数据所在的段中下降地比较慢，所以这些数据段将会在有一定碎片化的状态下停留一段时间。

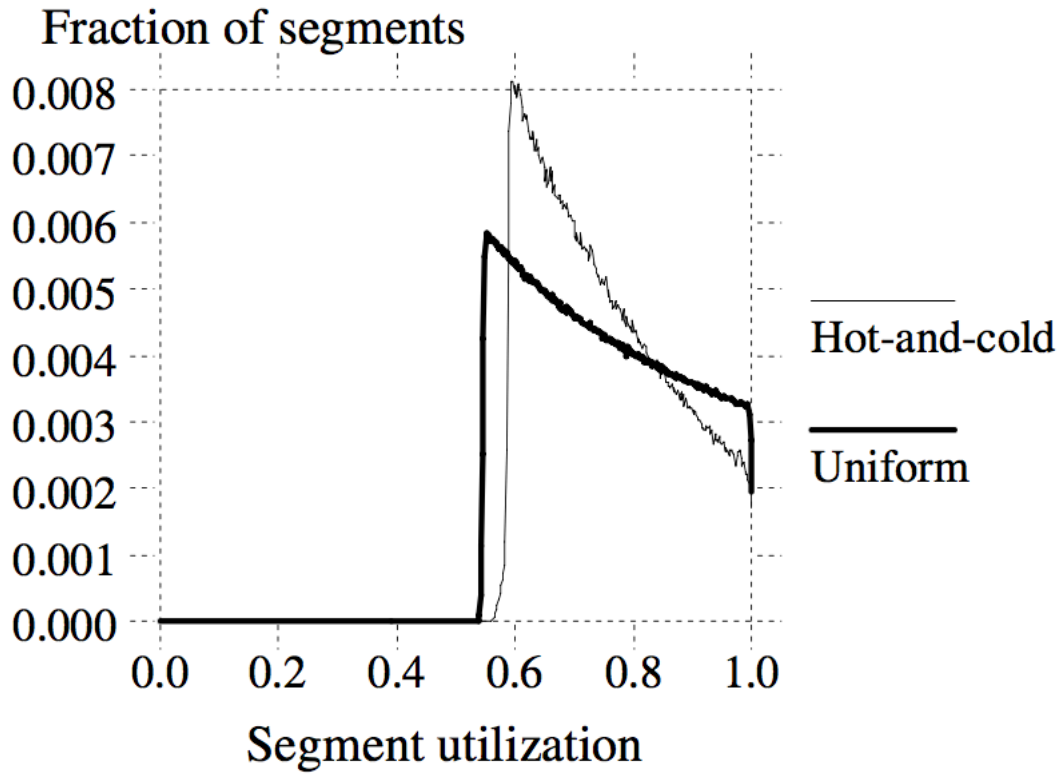

图5—在贪婪的整理策略在的段利用率分布（表达了不同算利用率的段在所有段中所占的比例）

在发现了这些特点之后，我们意识到冷数据段和热数据段应该被不一样地处理，在冷数据段中的空余空间应该比热数据段中的空余空间更有价值，因为冷数据段的空余空间将会在比较长的时间之内保持未使用的状态。换句话来说，当系统发现因为一个空闲的数据块是来自冷数据段，这些数据段将会被保持很长的时间，直到这些冷数据足够碎片化，才会进行整理。

相反，对于热数据段进行整理的收益是比较小的，因为数据块的状态会很快变化，空余空间需要被不断地重新整理；系统可能会延迟清理一段时间，让更多的数据块在这段时间之内发生变化。在段中的空余空间取决于这个段中数据的稳定性。不幸的是，这种稳定性是没有办法被预测的。假设一个数据在数据段中越长时间没有被使用就越不可能改变，那就可以使用数据的最后一次修改时间来估计数据的稳定性。

为了测试这个理论，我们在模拟中使用了新的选择策略来进行段的选择。这个策略将会根据段的整理收益和整理开销选择最适合的段来整理，力图选出效益最高的。收益包括两个部分：将要回收的可用空间量以及该空间可能保持空闲的时间量。可用空间量就是1-u，u就是之前提到的段利用率。我们使用段中数据块的最近修改时间来估计这个空间会在多长时间内保持空闲。对于整理这个空闲空间的收益就是这两个部分的乘积。而整个空间的清理开销就是1+u（需要一个单位的开销从段中读入数据，u个单位的开销用来写回数据）。将这些因素全部组合起来，我们就可以得到效益

$$\frac{\text{benefit}}{\text{cost}} = \frac{\text{free space generated} * \text{age of data}}{\text{cost}} = \frac{(1-u)*\text{age}}{1+u}$$

我们把这种策略叫做效益最佳策略（cost-benefit policy）；它使得冷数据段可以在相比热数据段更高的段利用率下被清理。

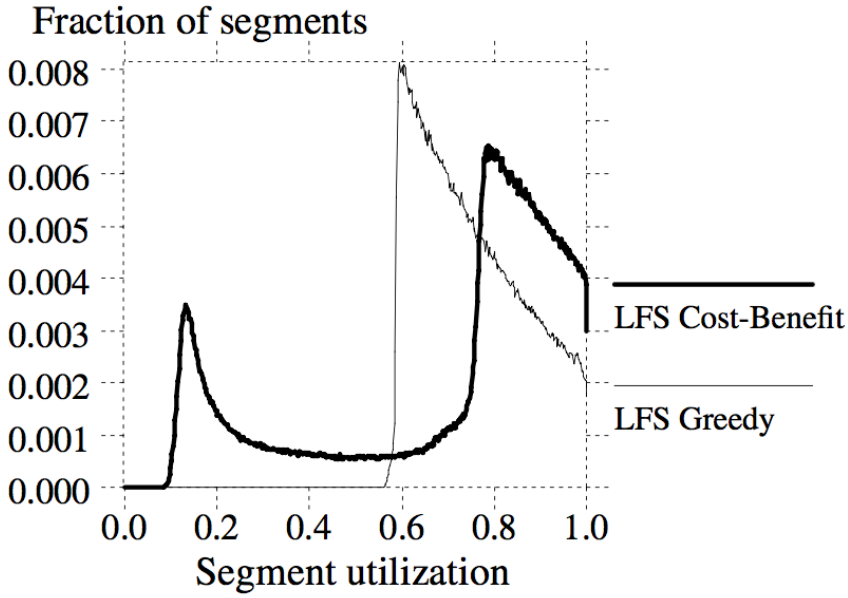

图6—在最佳效益策略下的算段利用率分布

我们重新在不均匀访问的模型下访问了整个模拟的过程，并且加入了效益最佳策略和数据块的修改时间排序。正如图6我们所能看到的，效益最佳的策略产生了我们所期望的双峰分布。这种整理策略将会清理段利用率为75%的冷数据段，但是对于热数据段来说这种策略会等待段利用率到达15%的时候才进行整理工作。因为90%的写入都是针对热数据的，大多数的被整理的段都是热数据段。图7展现出效益最小策略将会在相对贪婪策略上有超过50%的性能提升，即使在相对较高的磁盘容量利用率下，日志文件系统也能超过Unix FFS。我们也模拟不同程度的局部性访问，并且发现这种策略随着访问的不断增加局部性也会越来越好。

在模拟中的经验说服我们在Sprite LFS中我们要去实现效益最佳的策略。在章节5.2中，在文件系统的实际使用中，Sprite LFS的表现会比图7更好。

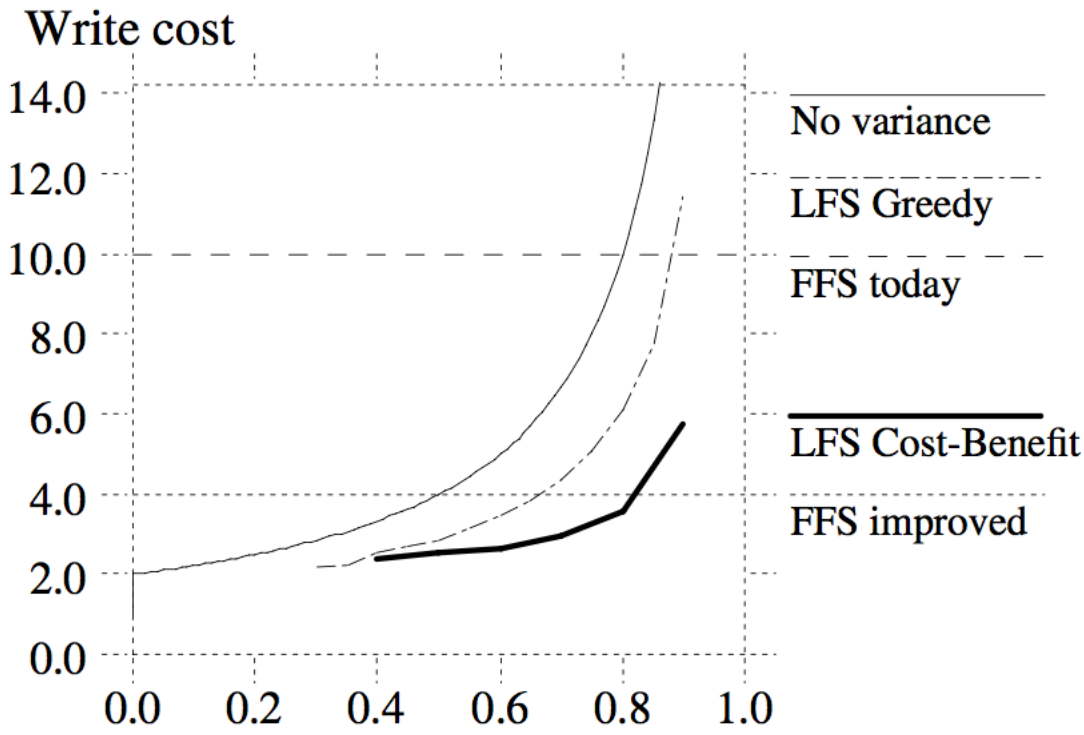

图7—在效益最佳策略下的写开销

3.6、段使用表

为了支持效益最佳的整理策略，Sprite LFS维护了一个叫做段使用表（segment usage table）的数据结构。对于每一个段，这个表格记录了每一个段的活跃（live）数据的大小，以及任何数据块的最后修改时间。这两个数据对于段整理器选择要整理的数据段是非常重要的。这些值在写入段

时初始设置，当文件被删除或块被覆盖时，有用（live）字节的数量越来越少。如果有有用字节的数量降到零，那么该段可以重复使用而不需要清洗。段使用表的数据块被写入日志中的，这些块的地址被存储在上文提到的checkpoint region中（在章节4中有细节的讲解）。

为了给有用（live）数据块的最近修改时间进行排序，一个段中最近修改文件的修改时间会被不断记录在段摘要信息中。目前，Sprite LFS并不保留文件中每个块的修改时间，它会为整个文件保留一个修改时间。对于未完整修改的文件，此估算值将不正确。我们计划修改段摘要信息以包含每个块的修改时间。

4、崩溃恢复

当一个系统发生了崩溃，最后在磁盘中进行的一些操作将会变成无一致性的状态（例如，新文件可能已被写入而不写入包含该文件的目录）；在重新提供的过程中，操作系统必须检查这些操作来纠正任何不一致状态。在传统的没有日志的Unix文件系统中，系统不能确定最后一次修改在哪里发生的，所以它扫描所有位于磁盘上的元数据结构来恢复一致性。这些扫描的开销非常高（在典型的配置上，需要几十分钟），当存储系统不断扩充，这个开销会越来越大。

在一个日志文件系统中，对于一盘的最后修改的位置是比较容易发现的：他就在日志的尾部。所以从崩溃中恢复是非常快的，日志的这种好处是众所周知的，并已被用于数据库系统和其他文件系统。像许多其他日志记录系统一样，Sprite LFS采用双管齐下的方法来恢复：定义文件系统一致状态的检查点，以及用于恢复自上一个检查点以来写入的信息的前滚（和数据库的恢复策略类似）。

4.1、检查点

检查点是一个在日志中的位置，在位置上文件系统的结构是一致并且完整的。Sprite LFS使用一个有两个步骤的方法来创建一个检查点。首先，它必须将所有的修改信息写到日志中，包括文件块，间接数据块，inode，node map以及段利用表。然后，它将checkpoint region写入磁盘上的特定固

定位置。检查点区域包含inode映射和段使用表中所有块的地址，加上当前时间和指向最后一个段的指针。

在重启的过程中，Sprite LFS读取checkpoint region并且使用这些数据来恢复在它的主存数据结构（很多元数据都是放在主存中，并且并不是永远都会持久化的磁盘中，所以这个持久化的工作是定期的）。为了处理在检查点操作时发生的崩溃，一般都会设定两个checkpoint region，checkpoint交替在它们之间进行操作。检查点时间位于检查点区域的最后一个块中，因此如果检查点失败，则不会更新时间。在重新引导期间，系统将读取两个检查点区域，并使用最近一次检查点区域。

Sprite LFS定期执行检查点，以及卸载文件系统或关闭系统。检查点之间的长时间间隔减少了编写检查点的开销，但增加了恢复期间需要前滚的时间;短的检查点间隔提高了恢复时间，但增加了正常操作的成本。 Sprite LFS当前使用30秒的检查点间隔，可能太短。定期检查点的替代方法是在给定数量的新数据写入日志之后执行检查点；这将会降低文件系统在负载较小的时候的检查点开销。

4.2、前滚（Roll-forward）

原则上，只要简单读取最新的检查点区域并在丢弃日志中的检查点之后任何数据，就可以安全地在崩溃后重新启动。这会导致瞬间恢复，但自从上一个检查点以来写入的任何数据都将丢失。为了恢复尽可能多的信息，Sprite LFS扫描上一个检查点之后写入的日志段。这个操作被称为前滚。

在前滚期间，Sprite LFS使用段摘要块中的信息来恢复最近写入的文件数据。当一个段摘要块指示还有一个新的inode的时候，Sprite LFS就会更新在checkpoint的inode map，从而inode map将会引用一个新的inode。这个过程将自动把文件的新数据块纳入到恢复之后的文件系统中。如果发现文件的数据块没有文件inode的新副本，则前滚代码假定磁盘上新文件的版本不完整，并忽略新数据块。

前滚程序的代码也要调整从检查点读到的段利用表的段利用率。在检查点之后写入的段利用率被设定为0；它们必须被重新调整来反应前滚前滚之后文件系统的真实状况。那些在检查点之前的，比较老的段的段利用率也

必须重新调整在反应最近的状况（这些都可以通过查看日志中的新inode来处理）。

前滚的最后一个问题是如何恢复目录条目和inode之间的一致性。每个inode包含引用该inode的目录条目数的计数;当计数下降到零时，文件被删除。不幸的是，当一个inode已经被写入到一个新的引用计数的日志，而包含相应的目录条目的块还没有被写入，或者反之亦然时，在这个过程中可能就会发生崩溃。

为了恢复目录和inode之间的一致性，Sprite LFS会在每个目录更改的日志中输出一个特殊的记录。这条记录包括操作码（可以代表create，link，rename，或者取消链接），目录条目的位置，目录条目的内容，以及条目中心inode的新引用计数。这些目录被集体叫做前文所提到的目录操作日志（directory operation log）；Sprite LFS保证了每个目录操作日志条目出现在相应目录块或inode之前的日志中。（目录操作日志应该是对目录修改的时候第一个进行的步骤）。

在前滚的过程中，目录操作日志用来保证目录条目和inode之间的一致性：如果一个日志条目存在，但是inode和目录块都没有被写入，那么前滚过程就会就会更新目录块和inode来完成这个操作。前滚操作可能会导致条目被添加到目录或从目录中删除，并且要更新的文件inode上的引用计数。恢复程序会将更改后的目录，inode，inode map和段使用表所在的数据块添加到日志中，并写入新的检查点区域以包含它们。唯一不能完成的操作是创建一个新的文件，inode永远不会被写入;在这种情况下，目录条目将被删除。除了其他功能之外，目录日志还可以轻松地提供原子重命名操作。

目录操作日志和检查点之间的交互将额外的同步问题引入了Sprite LFS。特别是，每个检查点必须相当于目录操作日志与日志中的inode和目录块一致的状态。这需要额外的同步，以防止在写入检查点时修改目录。

5、Sprite LFS的体验

从这个章节开始，日志文件系统就没有什么干货了，我们将简短地介绍Sprite LFS的测试结果。

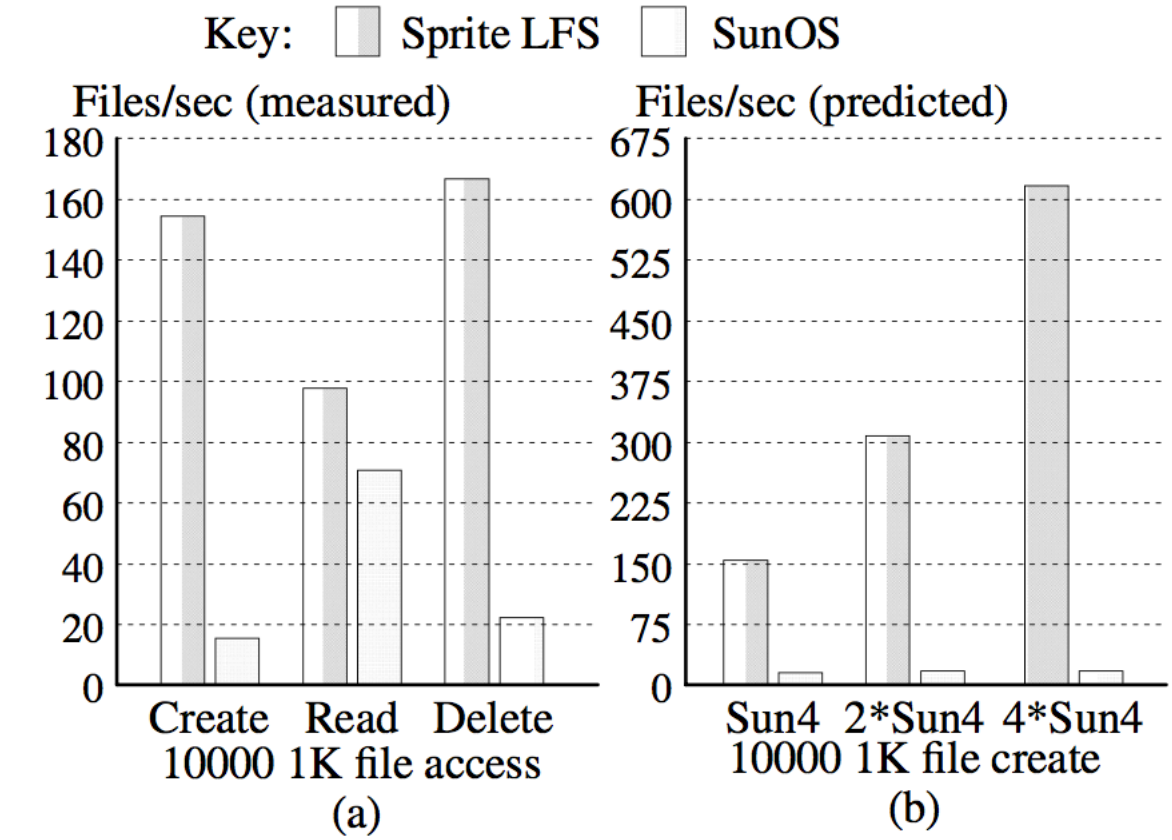

图8—在Sprite LFS与SunOS下的小文件性能

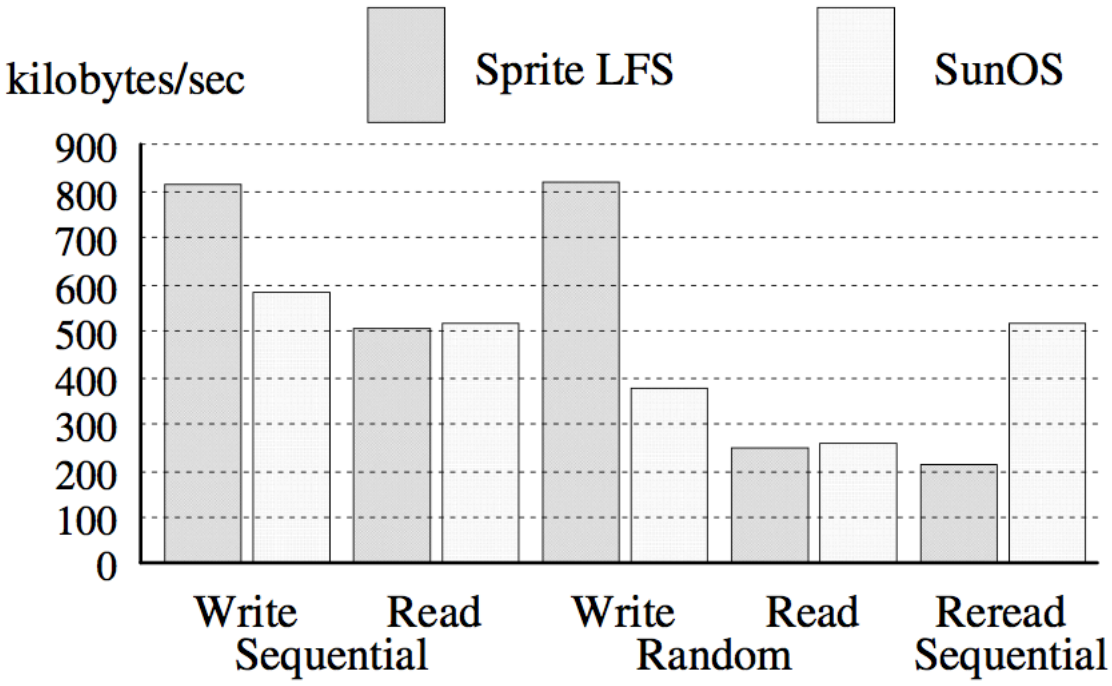

图9—Sprite LFS与SunOS大文件写入的性能

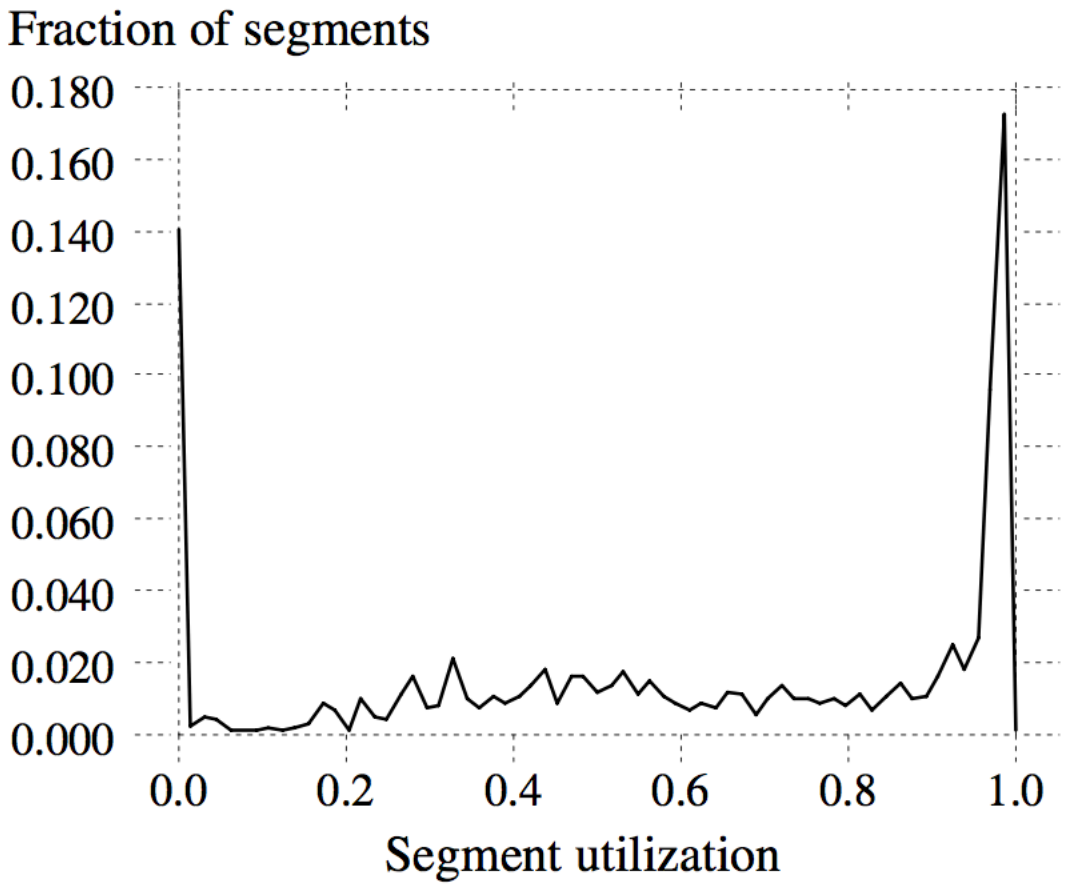

图10—段利用率的实际测试

6、相关工作

日志结构的文件系统概念和Sprite LFS设计借鉴了许多不同的存储管理系统的想法。类似于日志文件系统的观点已经出现很多次了。

Sprite LFS中使用的段清理方法的作用类似于为编程语言开发的清理垃圾收集器。在Sprite LFS中整理的块的效益最佳段选择侧列和块的最近修改时间排序将文件分割成几代，就像垃圾收集方案一样。这些垃圾收集方案和Sprite LFS之间的显着区别在于，在分代垃圾收集器中有效的随机访问是可能的，而顺序访问对于在文件系统中实现高性能是必要的。此外，Sprite LFS可以利用这样一个事实，即块一次最多只能属于一个文件，使用比用于编程语言的系统更简单的算法来识别垃圾。

Sprite LFS中使用的日志记录方案与数据库系统中首创的方案类似。几乎所有的数据库系统都使用预写式日志记录来实现崩溃恢复和高性能，但与Sprite LFS在使用日志方面有所不同。 Sprite LFS和数据库
系统将日志视为关于磁盘上数据状态的最新“真相”。主要区别在于数据库系统不使用日志作为数据的最终存储库：为此目的保留单独的数据区域。这些数据库系统的独立数据区域意味着它们不需要Sprite LFS的段清理机制来回收日志空间。日志在数据库系统中占用的空间可以在记录的更改写入其最终位置时回收。由于所有读取请求都是从数据区域处理的，所以可以大大压缩日志而不会影响读取性能。通常只有更改的字节被写入数据库日志而不是像Sprite LFS那样写入整个块。
检查点的Sprite LFS崩溃恢复机制和使用“重做日志”前滚的类似于数据库系统和对象库中使用的技术。 Sprite LFS中的实现被简化了，因为日志是数据的最终归属。 Sprite LFS恢复不是将操作重做到单独的数据副本，而是确保索引指向日志中数据的最新副本。
在文件缓存中收集数据并以大量写入磁盘的方式类似于数据库系统中的组提交和主内存数据库系统中使用的技术。

7、结论

日志结构文件系统的基本原理很简单：在主内存中的文件缓存中收集大量新数据，然后将数据写入磁盘中的一个大型I / O中，这些I / O可以使用所有的磁盘的带宽。实现这个想法很复杂，因为需要在磁盘上保留大量的空闲区域，但是我们的仿真分析和Sprite LFS的经验都表明，基于成本和收益的简单政策，可以实现低清洁开销。尽管我们开发了一个日志结构文件系统来支持具有许多小文件的工作负载，但这种方法对于大文件访问也非常有效。尤其是，对于整个创建和删除的非常大的文件来说，基本上没有清理开销。
底线是日志结构的文件系统可以比现有的文件系统更高效地使用磁盘。在I / O限制再次威胁计算机系统的可扩展性之前，这应该可以利用几代更快的处理器。