LFS:The Design and Implementation of a Log-Structured File System

`HDD` `FileSystem` `Log-Structured`

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

背景

80年代，CPU处理速度得到极大提升而磁盘的访问速度却提升很慢（ “Memory/Storage Wall” ），为了降低磁盘IO与CPU处理速度不匹配所带来的影响，本文提出一种日志结构的文件系统，充分发挥磁盘顺序读写的高效率。

1990's文件系统设计

> 当时，文件系统的设计受**技术发展**和**工作负载**两方面的影响。

从技术发展方面来说，处理器的效率以指数效率提升，与主存、磁盘访问呈现出不平衡的发展趋势。磁盘的性能主要体现在传输带宽和访问时间上，传输带宽的短板可以通过使用磁盘阵列和并行磁头[5]来弥补，但访问时间受机械运动的限制，难以从制造上进行优化。主存的容量在今年来也得到了很大发展，几乎也以指数增长，从而使大容量的缓存成为可能。大容量缓存，一方面能很大几率命中负载中大量的读操作，另一方面，零散的写操作也可以在缓存中进行聚集。
从工作负载方面来说，一种最难处理的是办公和工程环境下的文件系统负载，特点是存在大量的小文件（KB）读写，带来大量的磁盘随机I/O操作，文件系统元数据操作成为瓶颈；另外，在超算环境下，大文件的顺序访问较为常见，许多工作已经能够有效保证大文件在磁盘上的顺序化布局，这种情况下，I/O性能受限于I/O带宽和主存子系统而不是文件的数据布局。
当时，文件系统（1）数据的组织和布局方式带来大量小的磁盘访问 （2）倾向于使用同步写，这种方式虽然简化了灾后恢复逻辑，却也损害了写性能。
本文的日志型文件系统设计，着重于提高小文件访问的效率，特别是写效率。设计的前提在于文件能够缓存于较大的主存缓存中，最大的挑战则在于如何保证随时都有较大的连续的空闲空间以供数据的日志型追加写入。比较对象是Berkeley Unix fast file system(Unix FFS)。

日志型文件系统（LFS）设计

> **数据的组织**和**空闲空间管理**是LFS设计的两大核心部分

1. 数据的组织

- 与FFS相同，使用inode记录文件的属性，如类型、所有者、权限等，并且记录了10个文件数据块的地址，当文件较大时，使用间接块来记录文件数据块地址。FFS中每个inode都保存在固定的位置（文件更新为原地更新），根据文件的识别号简单计算可得。而LFS中，采用Inode map来记录每个inode的地址，并且通过文件识别号，可从中检索到对应的文件inode磁盘地址。Inode map也会以日志形式持久化到磁盘上，checkpoint区域会记录inode map持久化的数据地址。
- 磁盘分为多个固定大小的盘区（Extents）,称为片段（Segments）。当日志操作写满一个片段时，由于数据的删除和更新操作，片段上会有一些数据已经失效（随机分布），此时有两种策略：threading和copying。threading的方式是跳过活跃的数据进行写操作；copy则是将活跃数据拷贝出去。对threading而言，大的连续空间写操作性能会受到影响；copying则会带来一些额外的读写开销。本文中的设计将二者进行结合，片段内采用copying方法，片段间采取threading的方法。

Data structure	Purpose	Location	Section
Inode	Locates blocks of file, holds protection bits, modify time, etc.	Log	3.1
Inode map	Locates position of inode in log, holds time of last access plus version number.	Log	3.1
Indirect block	Locates blocks of large files.	Log	3.1
Segment summary	Identifies contents of segment (file number and offset for each block).	Log	3.2
Segment usage table	Counts live bytes still left in segments, stores last write time for data in segments.	Log	3.6
Superblock	Holds static configuration information such as number of segments and segment size.	Fixed	None
Checkpoint region	Locates blocks of inode map and segment usage table, identifies last checkpoint in log.	Fixed	4.1
Directory change log	Records directory operations to maintain consistency of reference counts in inodes.	Log	4.2

Table 1 — Summary of the major data structures stored on disk by Sprite LFS.
For each data structure the table indicates the purpose served by the data structure in Sprite LFS. The table also indicates whether the data structure is stored in the log or at a fixed position on disk and where in the paper the data structure is discussed in detail. Inodes, indirect blocks, and superblocks are similar to the Unix FFS data structures with the same names. Note that Sprite LFS contains neither a bitmap nor a free list.

2. 空闲空间管理

> 主要介绍片段清理机制，这是保证有大块空闲空间的关键。其主要步骤包括：（1）将部分片段数据读进内存 （2）识别活跃数据 （3）将活跃数据写回到小部分的干净片段中。需要解决的问题：（a）片段清理何时执行？设置阈值。 （b）每次清理多少片段？50-100.（c）什么样的片段需要清理？ （d）清理时，拷贝出来的活跃数据块应该如何组织？本文主要讨论c,d两方面的问题。

- 活跃数据识别。片段中保存一个片段总结块（Segment summary block），其中记录着如，文件号和文件数据块号等信息。通过查询文件Inode信息可以确定数据是否是最新活跃的数据。本文还采用版本号来优化识别的效率。
- 片段清理机制的讨论
  采用write-cost来形式化表征清理机制带来的开销，其表达式及其曲线变化如下。获得高性能的关键在于使磁盘空间的u呈现出驼峰分布，即对于大量的段，尽可能处于满的状态（有效数据多）或空（有效数据少）的状态。

> $write\_cost = \frac{total\ bytes\ read\ and\ written}{new\ data\ written} = \frac{read\ segs + writelive + write\ new}{new\ data\ written} = \frac{N + N*u + N*(1-u)}{N*(1-u)} = \frac{2}{1-u}$
> u为段内空间利用率，即活跃数据比例。

- 贪婪的清理策略，对u最小的段进行清理。对Uniform分布和Hot-and-cold分布的文件访问特性进行测试，结果发现，随着访问局部性的增加，write-cost反而增加。分析得知，在贪婪策略的作用下，大部分段的u值趋向于向中间分布，出现了大量冷数据段长期处于清理边缘，占据大量空闲块的情况。
- 基于代价的清理策略（cost-based polocy)。 通过分析可以认识到：冷热数段空闲空间价值要比热数据段的大；一旦一个冷数据段选作进行清理操作，这些冷数据仍然要保留很长时间。因此段中空闲空间的价值与数据的稳定性相关。这里引入age（段中最近被修改的块的修改时间，在段总结信息中保存）作为稳定性的度量。结果表明，这种策略能够对有局部性访问特征的情况有很大改善，并且出现了驼峰分布。为了支持这个策略，使用Segment usage table记录段中活跃数的数量和段的age。

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

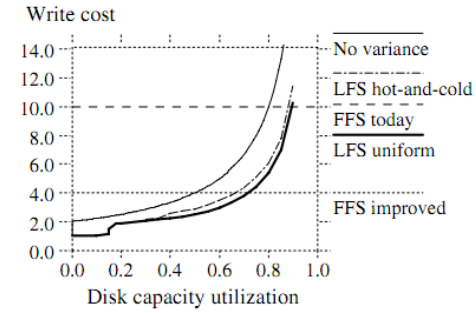

Figure 4 — Initial simulation results.

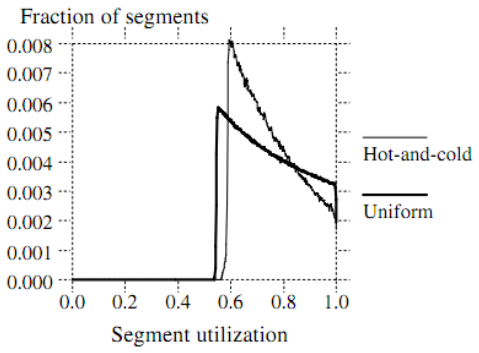

Figure 5 — Segment utilization distributions with greedy cleaner.

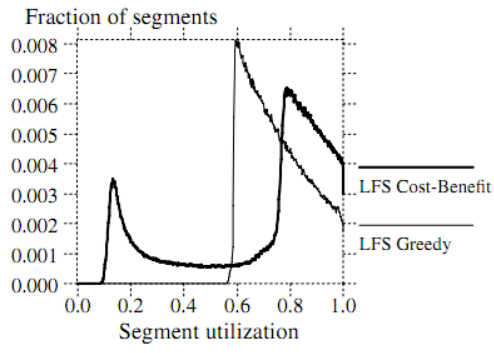

Figure 6 — Segment utilization distribution with cost-benefit policy.

3. 灾后恢复

> 对传统的文件系统来说，需要扫描全盘来恢复系统状态。对于LFS来说，扫描日志即可。LFS采取了两种方式来保证灾后恢复，**checkpoints**和**roll-forward**。

- Checkpoints。第一步将所有的修改写到日志中，包括文件数据块，间接块，inodes,inode map

块，segment usage table块等。第二步在特定的checkpoint区域记录块的地址，如inode map中所有的块等，还有当前时间和指向最后被写的段的指针。为了处理checkpoint过程中出现的故障，LFS保留两个checkpoint区域以及两个checkpoint之间变化（Operation Alternate）。

- Roll-forward。恢复时，除了恢复到上一次checkpoint的状态，还可以扫描checkpoint时间之后的日志，进行进一步的恢复。

4. 测试与结论

MIcro-benchmarks, 与 SunOS 4.0.3的UnixFFS在大文件，小文件读写下进行比较。LFS在写性能上有显著的优势，并且故障恢复开销也更小。

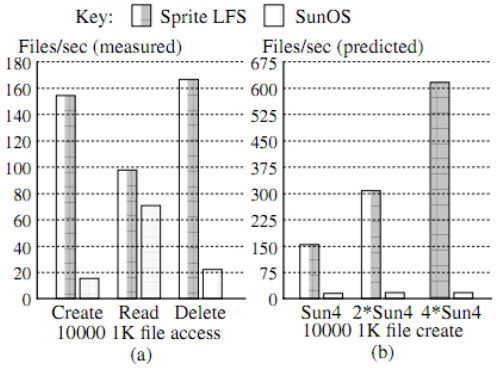

Figure 8 — Small-file performance under Sprite LFS and SunOS.

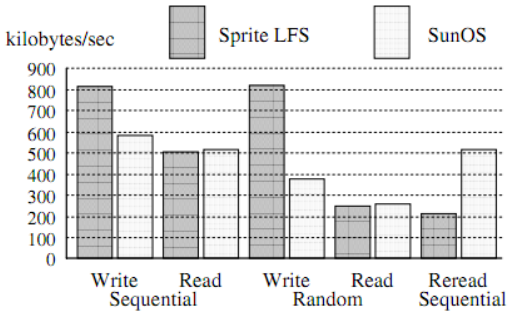

Figure 9 — Large-file performance under Sprite LFS and SunOS.

总结

总体来说，LFS通过在缓存中进行数据的聚集，然后使用大IO和日志Append写的方式用满磁盘的带宽，使得磁盘的使用效率有数量级的提高；其主要问题在于Cleaning操作具有一定的开销。

Write cost in Sprite LFS file systems

File system	Disk Size	Avg File Size	Avg Write Traffic	In Use	Segments Cleaned	Segments Empty	*u* Avg	Write Cost
/user6	1280 MB	23.5 KB	3.2 MB/hour	75%	10732	69%	.133	1.4
/pcs	990 MB	10.5 KB	2.1 MB/hour	63%	22689	52%	.137	1.6
/src/kernel	1280 MB	37.5 KB	4.2 MB/hour	72%	16975	83%	.122	1.2
/tmp	264 MB	28.9 KB	1.7 MB/hour	11%	2871	78%	.130	1.3
/swap2	309 MB	68.1 KB	13.3 MB/hour	65%	4701	66%	.535	1.6

Table 2 - Segment cleaning statistics and write costs for production file systems.
For each Sprite LFS file system the table lists the disk size, the average file size, the average daily write traffic rate, the average disk capacity utilization, the total number of segments cleaned over a four-month period, the fraction of the segments that were empty when cleaned, the average utilization of the non-empty segments that were cleaned, and the overall write cost for the period of the measurements. These write cost figures imply that the cleaning overhead limits the long-term write performance to about 70% of the maximum sequential write bandwidth.