The Design and Implementation of a Log-Structured File System.

M. Rosenblum, J. Ousterhout.
ACM Trans. Computer Systems,
vol. 10, no. 1, Feb 1992, pp. 26-52.

Dealing with Disks 2

- Today's troubles:
  - Disk seeks are evil
    - In the paper: 1.3MB/sec disk, 17.5ms seek
    - Today: 30x faster xfer, 4x better seek. Ouch!
  - Synchronous writes in conventional filesystems are evil
    - Application blocks
    - Most sync writes involve ... seeks! Possibly many.

Common Workload Patterns

- Back to our old 90/10 rules of thumb:
  - Ganger: 79% of files < 8KB
  - Baker: 80% accesses to files < 10KB
  - (But most bytes on disk in large files!)
    - e.g., 21 files on my laptop occupy 23G / 93G =1/4!
    - Of course, single largest file is 6G of 15-712 lecture video. :) == 6.5% of hard drive for one file...
  - Suggests many ways to optimize

Write vs Read Oriented Layout

- Importance of disk write BW (think about RAID - small writes are slower...)
  - Successful read caching and prefetching (false - paper mis-predicted)
    - Unfortunately, larger DRAM offset by larger files, bigger disks & their metadata structures, larger app VM working sets
    - But write order good for read unless seq reads after random write
  - Synchronous writing for recovery is bad (true)
    - Part answers: battery backup writeback caches for power fail, and recoverable main memory structures simple crashes
- Use of transaction logging to avoid FSCK
  - Recovery of inflight operations between checkpoints
  - whole-disk fsck gets increasingly bad as MAD grows > Tx rate
- LFS led into new era of disk layout (hybrid log+mapped)
  - NetApp WAFL industry leader in distr'd file systems

Data & Metadata Layout

- Updated structures written to end of log
  - Buffer as long as possible to minimize seeks
  - Compare to FFS seek per 2 file inode writes, 1 file data write, 1 directory inode write, 1 directory data write (& free bitmap writes)

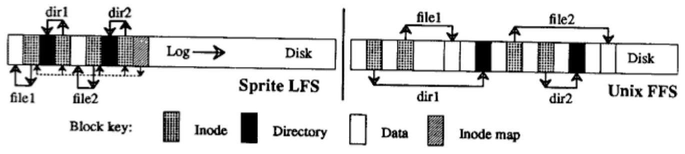

Fig. 1. A comparison between Sprite LFS and Unix FFS. This example shows the modified disk blocks written by Sprite LFS and Unix FFS when creating two single-block files named dir1/file1 and dir2/file2. Each system must write new data blocks and inodes for file1 and file2, plus new data blocks and inodes for the containing directories. Unix FFS requires ten nonsequential writes for the new information (the inodes for the new files are each written twice to ease recovery from crashes), while Sprite LFS performs the operations in a single large write. The same number of disk accesses will be required to read the files in the two systems. Sprite LFS also writes out new inode map blocks to record the new inode locations

Potential Performance

- Reducing seeks goes long way to make file system faster
  - No cleaning done in benchmark

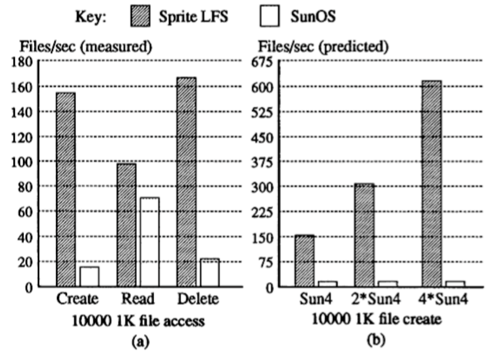

Fig. 8. Small-file performance under Sprite LFS and SunOS. (a) measures a benchmark that created 10000 one-kilobyte files, then read them back in the same order as created, then deleted them. Speed is measured by the number of files per second for each operation on the two file systems. The logging approach in Sprite LFS provides an order-of-magnitude speedup for creation and deletion. (b) estimates the performance of each system for creating files on faster computers with the same disk. In SunOS the disk was 85% saturated in (a), so faster processors will not improve performance much. In Sprite LFS the disk was only 17% saturated in (a) while the CPU was 100% utilized; as a consequence I/O performance will scale with CPU speed.

Cleaning the log

- But space freed by file deletes is needed

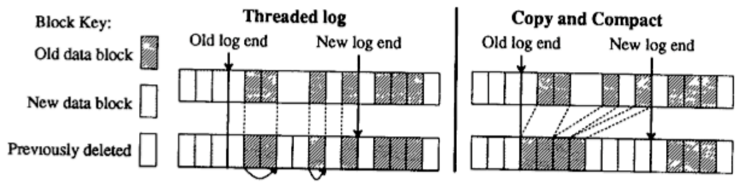

Fig. 2. Possible free space management solutions for log-structured file systems. In a log-structured file system, free space for the log can be generated either by copying the old blocks or by threading the log around the old blocks. The left side of the figure shows the threaded log approach where the log skips over the active blocks and overwrites blocks of files that have been deleted or overwritten. Pointers between the blocks of the log are maintained so that the log can be followed during crash recovery The right side of the figure shows the copying scheme where log space is generated by reading the section of disk after the end of the log and rewriting the active blocks of that section along with the new data into the newly generated space.

Cleaning con't

- Cleaning needs block# to file/offset map
  - FFS bitmap can't, so add segment summary
  - Liveness (easy in bitmap) done by testing block -> inode -> block pointer cycle
- Divide into segments & thread log thru segments
- Order: clean most free space first (greedy)?
  - But old segments (unlikely to see deletes) don't give up their space for long time; recently written segments, seeing lots of deletes, will see more soon
  - Cost-Benefit metric increases with time since last write and increases with more free space
    - Old segments cleaned at 15% free, new segments at 75%, if 5% of files get 90% of writes
    - Also, cleaning many segments, reorder to pack old files in different segments from new files

Greedy-Cost or Cost-Benefit

$$\text{write cost} = \frac{\text{total bytes read and written}}{\text{new data written}} = \frac{2}{1-u} \qquad \frac{\text{benefit}}{\text{cost}} = \frac{\text{free space generated*age of data}}{\text{cost}} = \frac{(1-u)\text{*age}}{1+u}.$$

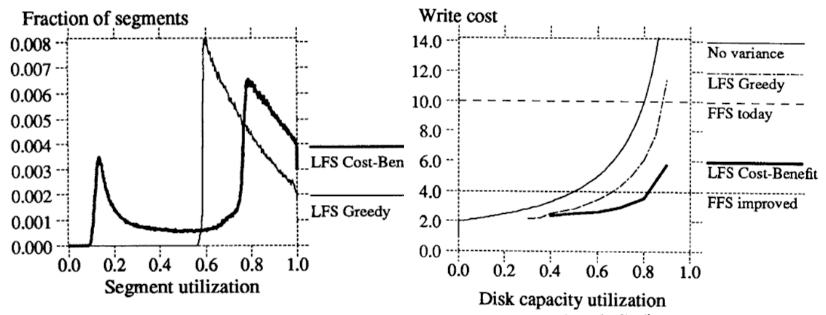

Fig. 6. Segment utilization distribution with cost-benefit policy. This figure shows the distribution of segment utilizations from the simulation of a hot-and-cold access pattern with 75% overall disk capacity utilization. The "LFS Cost-Benefit" curve shows the segment distribution occurring when the cost-benefit policy is used to select segments to clean and live blocks are grouped by age before being rewritten. Because of this bimodal segment distribution, most of the segments cleaned had utilizations around 15%. For comparison, the distribution produced by the greedy method selection policy is shown by the "LFS Greedy" curve reproduced from Figure 5.

Reconsidering Cleaning

- LFS uses continuous/threshold-drive cleaning to defragment
  - Emphasizes not cleaning "hot segments" too soon as they will see lots more deletes soon (wasting copies)
- Seltzer95 (Usenix95) disagrees
  - Fixed some old non-optimal FFS actions (update 1984 technology to 1995 -- LFS compared to weak target)
    - Clustered writes - group inodes + data, fewer seeks
  - Finds cleaning costs often outweigh benefits; background cleaning may defray cost to user workload
  - Very workload dependent -- benefits of LFS best in small file create loads
- NetApp WAFL doesn't clean automatically
  - Admin tool for defrag as needed (if needed)

Recovery after Crash

- To avoid fsck ("find" & test all metadata)
- Checkpoint "top of FS" to fixed location
  - Contains pointer to last written segment, stats
  - Double buffer checkpoint, use latest timestamp
    - Every 30sec (better: min(30sec, X MB written))
- Roll-forward log-segments after checkpoint
  - Embed write-ahead log for directory changes (create, link, unlink, rename) so interrupted log writing can correct directory/inode pointers/refcounts (except creates without inode making it to disk)

Data vs Metadata

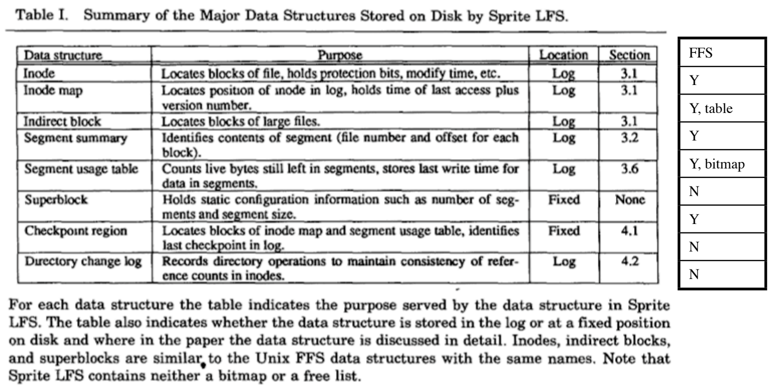

Table I. Summary of the Major Data Structures Stored on Disk by Sprite LFS.

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

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 or a free list.

- WAFL trick for floating inodes: one fixed inode points at file containing table of rest of inodes
  - Replaces inode map with this file's inode + arithmetic

Eval

- Modern "build & measure + simulate"
  - Excellent problem formulation & metrics
  - Big dependence on workload, but sample of workloads explored not wide
  - Key ideas about cleaning but benchmarks too often don't control/test cleaning
- Key idea: merge writebacks to reduce seeks
  - Cleaner theory was what they thought was key, but in fact it remains untrusted by many

Benchmarks Today

- Some people scale up MAB
- Postmark (mail server sim - lots of small writes)
- Netnews (tons of creates/deletes, files)
- Large file writes (easy to test)
- "Modern" MAB - untar/compile your favorite software package (emacs, ssh, ...)

The Modern World

- WAFL - "Write Anywhere Filesystem"
  - Write-ahead logging of NFS requests
  - Snapshots for consistency (batch up lots of requests, write out en masse periodically, snapshot)
  - Write data + inodes to any free block the disk head is near; update master pointers later
  - NVRAM to avoid need for synchronous writes ($$, but appliances are already $$$)

Modern, 2

- Journaling filesystems
  - Write (async or sync) metadata updates to a log in a big stream for consistency
  - Update "real" filesystem contents when appropriate
  - Duplicates some metadata writes
    - But lets you defer/batch seek-intensive ops, so worth it
  - Used in a lot of modern filesystems!

Modern 3: Soft Updates

- Instead of logging...
- ... Allow writes partially out-of-order, and defer many of them
  - Why defer? create/write/remove - ends up with no disk writes at all...
- How? Track dependencies between dir/inode/data writes (graph of ops)
  - Requires some complex rollback machinery
  - But works pretty well when you get it right