

Log-Structured File Systems

CS 416: Operating Systems Design

Department of Computer Science

Rutgers University

<http://www.cs.rutgers.edu/~vinodg/teaching/416/>

Motivation

Three observations:

Memory speeds are getting faster at an exponential rate

Disk access speeds are also increasing, but not as fast

Caches increasing in size

Implication

File read requests will most likely be satisfied from cache

Disk traffic will be dominated by writes

Why can't these be satisfied by cache as well?

Can we design a new file system optimized to work well under these observations?

Log-structured file system

Store data on disk as a log: a sequential structure aimed at eliminating all seeks.

Side benefit: better crash recovery.

Improves performance over Unix FFS by an order of magnitude for several workloads

Two factors influencing file system design

Disk Technology:

Raw disk bandwidth: Number of bytes/second that can be transferred to the disk:

Access time: Time taken to access a particular byte on disk. Includes time to look up location on disk and seek to that location.

Workloads:

Office workloads: Lots of writes to small files (few kilobytes). Random disk I/Os.

Scientific workloads: sequential access to large files.

LFS is optimized for office workloads, but works well for scientific workloads as well.

What's wrong with existing file systems?

Spread information around on disk

Unix FFS lays files out sequentially, but physically separates files

Inodes are stored separately from the actual contents of the file

Result: At least five separate disk I/Os to create a file on disk.

Less than 5% of the disk b/w is used when writing small files.

File systems perform synchronous writes

Application waits for write to complete

Even if data blocks written asynchronously, metadata writes are synchronous: Why?

LFS: main ideas

Main idea: Buffer a sequence of file system changes in a file cache. Write these all out to disk sequentially in a single operation

Everything is stored in this log

Data blocks, attributes, inodes, directories, etc.

Contrast to traditional UNIX file systems.

Main issues:

How to read data from disk?

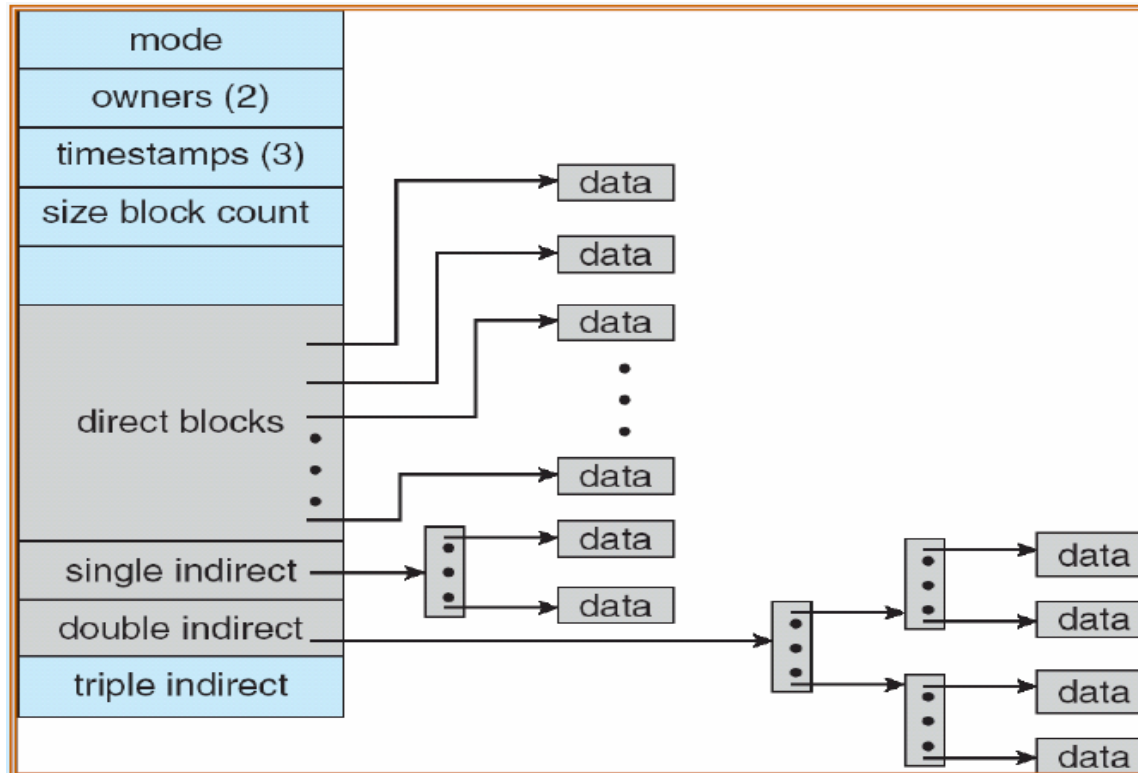
How to manage free space on disk?

Reading data from disk: FFS

How are reads accomplished on traditional Unix file systems?

Read inode

Follow pointers to data blocks or to indirect blocks



Reading data from disk: LFS

Problem: How to locate inodes – they're part of the log itself!

Solution: inode map – maintains the location of each inode.

inode map is itself divided into blocks that are written to the log

fixed checkpoint region on disk stores locations of all inode maps

Key to performance:

inode maps can fit in main memory → inode map lookups rarely require disk accesses

File layout in LFS versus Unix FFS

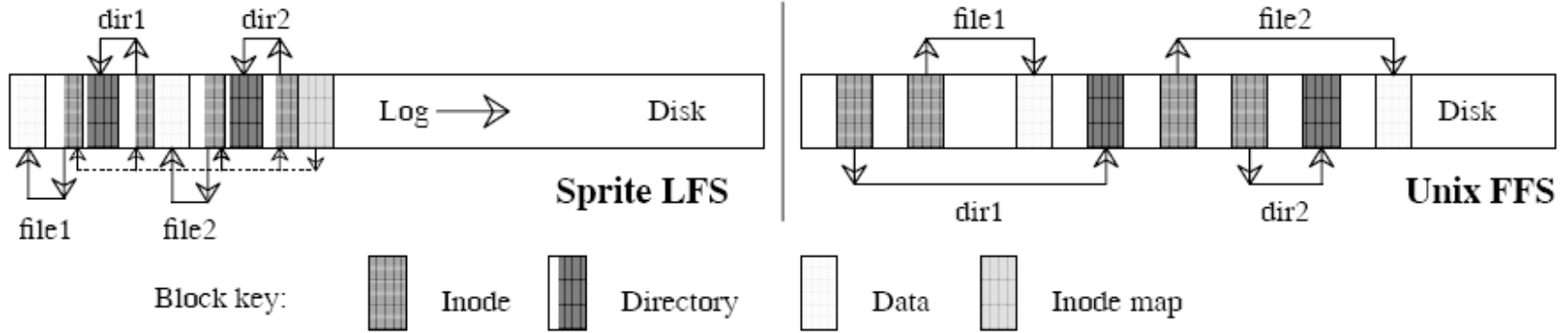


Figure 1 — A comparison between Sprite LFS and Unix FFS.

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 non-sequential 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.

Free space management

Problem: How to manage free space on disk?

Solution adopted by FFS: Bitmaps, and free-block lists

In LFS, there are two design options:

- Threading blocks in the log

- Copy and compact

Threading and copying

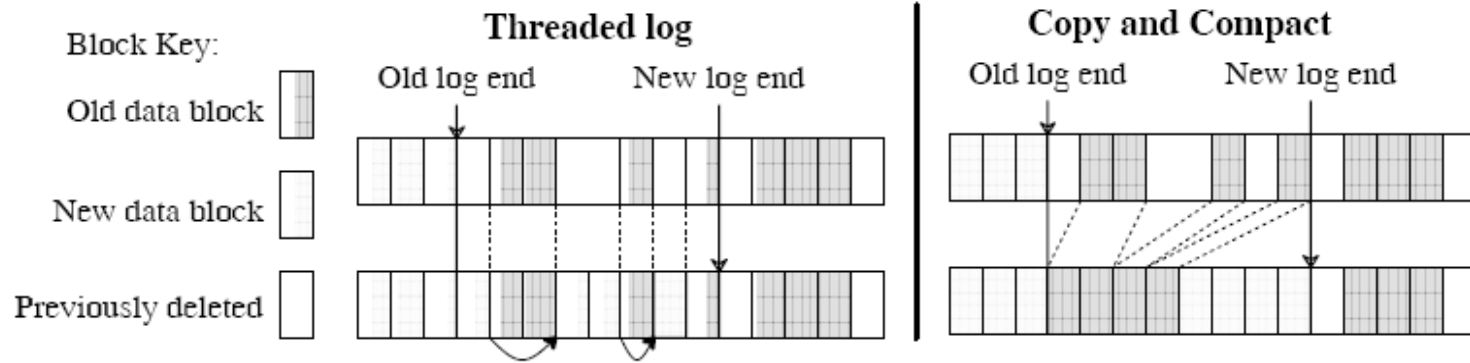


Figure 2 — Possible free space management solutions for log-structured file systems.

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.

Free space management: Segments

LFS uses a combination of threading and copy-and-compact

Organize disk into large, fixed-size extents called segments:

Segments written sequentially from beginning to end.

Log is threaded on a segment-by-segment basis

Perform copy-and-compact on a per-segment basis: Segment cleaning

Segment cleaning

Happens in three steps:

Read a number of segments into memory

Identify disk blocks containing live data

Write it back into a smaller number of segments.

Problem: How to identify which blocks are live? Which files do they belong to?

Use a **segment summary block** to store this information.

Segment cleaning issues

1. When should the segment cleaner execute?
2. How many segments should it clean at any given time?
3. Which segments should be cleaned?

Obvious choice → ones that are most fragmented.

Does this work well?

4. How should live blocks be grouped when they are written out?

Try to enhance locality of future reads.

Sort blocks by the time that they were last modified. Group blocks of similar age together: **age sort**.

Issue (3) turns out to be the most significant factor in determining performance

Cost-benefit analysis

Use a cost-benefit analysis to identify blocks to be cleaned

Write cost = Average amount of time that the disk is busy per byte of new data written, including cleaning overheads

$$\begin{aligned}\text{write cost} &= \frac{\text{total bytes read and written}}{\text{new data written}} \\ &= \frac{\text{read segs} + \text{write live} + \text{write new}}{\text{new data written}} \\ &= \frac{N + N*u + N*(1-u)}{N*(1-u)} = \frac{2}{1-u}\end{aligned}$$

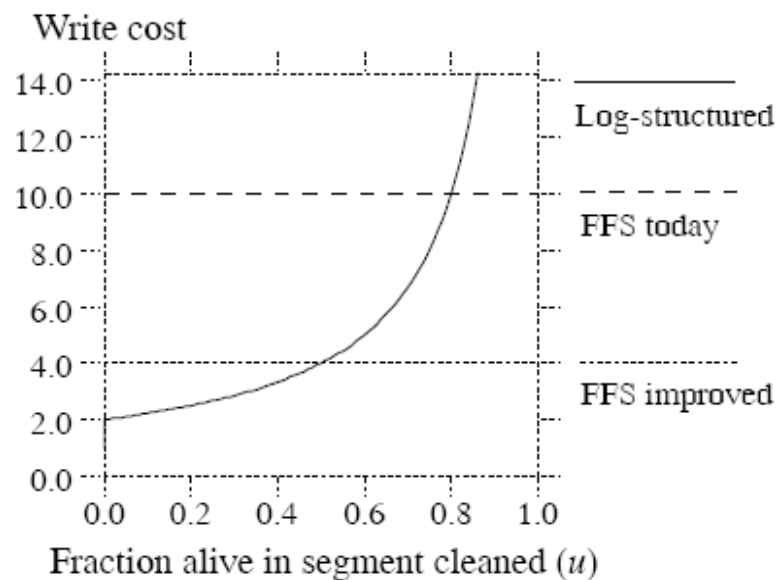


Figure 3 — Write cost as a function of u for small files.

In a log-structured file system, the write cost depends strongly on the utilization of the segments that are cleaned. The more live data in segments cleaned the more disk bandwidth that is needed for cleaning and not available for writing new data. The figure also shows two reference points: “FFS today”, which represents Unix FFS today, and “FFS improved”, which is our estimate of the best performance possible in an improved Unix FFS. Write cost for Unix FFS is not sensitive to the amount of disk space in use.

Implication: performance/utilization tradeoff

Low disk utilization → LFS works well, since segment cleaning is cheap.

High disk utilization → Storage costs reduced, but performance of LFS also degrades.

Similar performance tradeoffs with FFS

Which segment to choose?

LFS treats hot segments differently from cold segments

Hot segments: those that are frequently accessed.

Idea: Clean cold segments to reclaim space

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

Gives priority to older segments.

Intuition: Free space gained from a cold block can be kept for a longer time than free space gained from a hot block

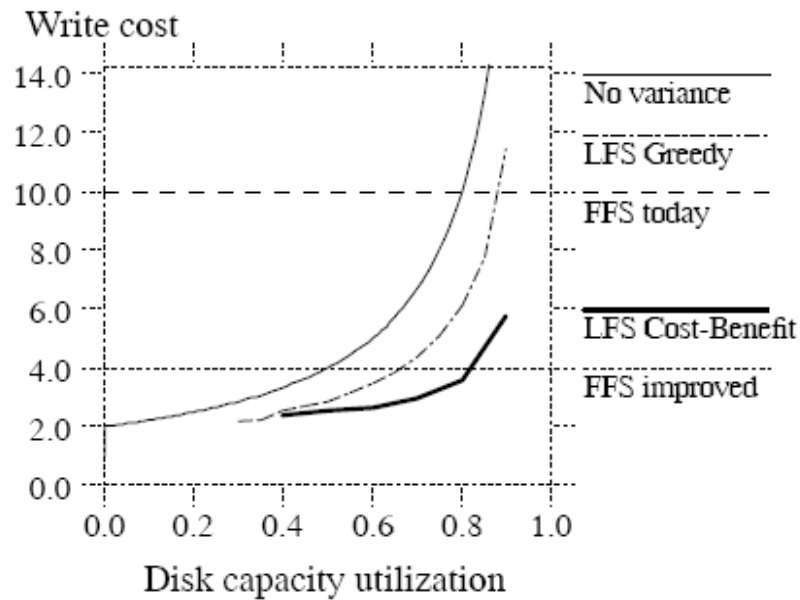


Figure 7 — Write cost, including cost-benefit policy.

This graph compares the write cost of the greedy policy with that of the cost-benefit policy for the hot-and-cold access pattern. The cost-benefit policy is substantially better than the greedy policy, particularly for disk capacity utilizations above 60%.

Performance of LFS

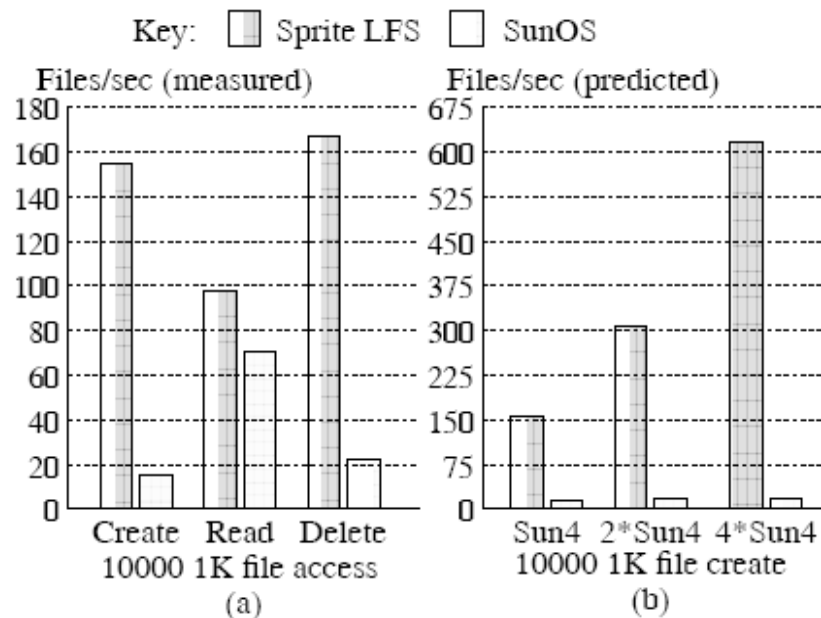


Figure 8 — Small-file performance under Sprite LFS and SunOS.

Figure (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. Figure (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.

Performance of LFS

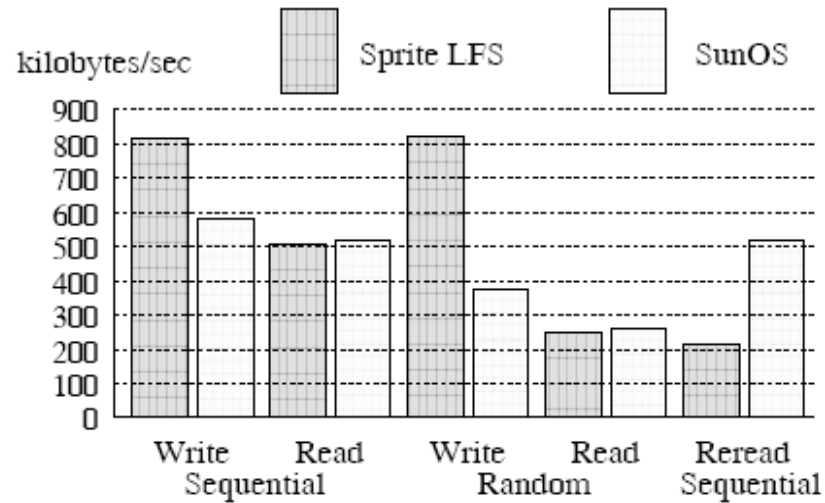


Figure 9 — Large-file performance under Sprite LFS and SunOS.

The figure shows the speed of a benchmark that creates a 100-Mbyte file with sequential writes, then reads the file back sequentially, then writes 100 Mbytes randomly to the existing file, then reads 100 Mbytes randomly from the file, and finally reads the file sequentially again. The bandwidth of each of the five phases is shown separately. Sprite LFS has a higher write bandwidth and the same read bandwidth as SunOS with the exception of sequential reading of a file that was written randomly.

Recovering from crashes

How to recover from a crash on FFS?

Utilities such as fsck potentially traverse entire disk.

LFS: Crash can be localized to the last few entries of the log!

Two concepts used for recovery:

Checkpoints

Roll-forward

Checkpoints

Position in the log where all data is consistent and complete

Use two phase process to create checkpoint.

Write out all modified data to the log

Write a checkpoint region to a fixed position on disk. Write all the address of blocks in the inode map, segment usage table, current time.

During reboot, read checkpoint region to initialize main-memory structures.

Corner case: what if there's a crash during the checkpointing process? 😊

Roll forward

One option during restart: just discard any data written after the last checkpoint. Instantaneous recovery, but data since last checkpoint is lost.

Try to recover as much information as possible after last checkpoint: roll forward.

Use segment summary blocks to recover recently-written file data.

If you find an inode, update inode map.

Adjust segment utilization data.