The Design and Implementation of a Log-Structured File System

Rosenblum, M., & Ousterhout, J. K., **1992 ACM Transactions on Computer Systems**

2024. 07. 10

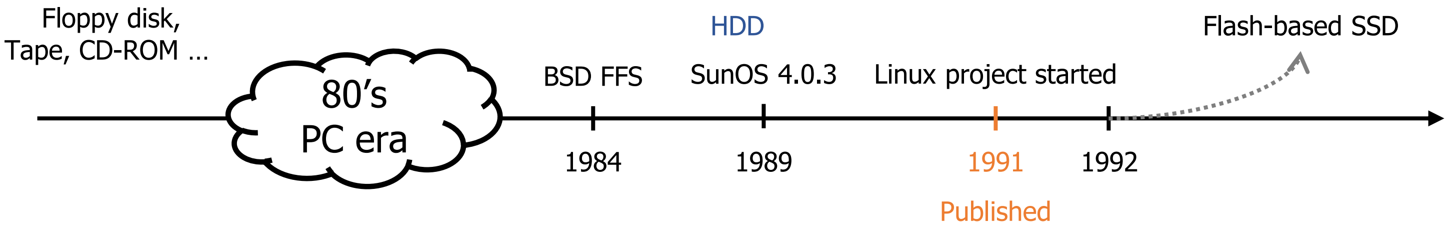
Presentation by Nakyeong Kim

nkkim@dankook.ac.kr

Contents

1. Introduction

Background

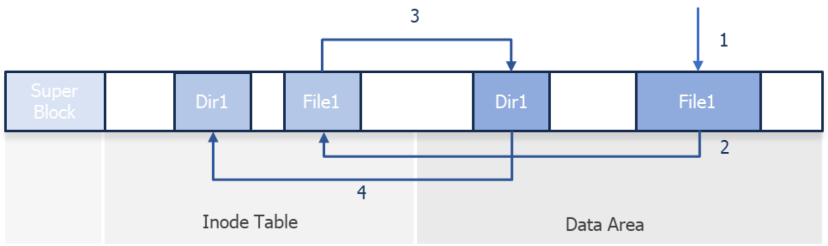


- Main memories are dramatically grown up
  - Large file caches can absorb read requests
  - Disk traffic will become dominated by **writes**
- Disk's random I/O performance is very poor
  - Decreasing mechanical cost is limited
- Existing file systems(e.g., FFS) spread information in disk: many small accesses
  - Bandwidth usage for new data is less than 5%
  - Metadata is written synchronously

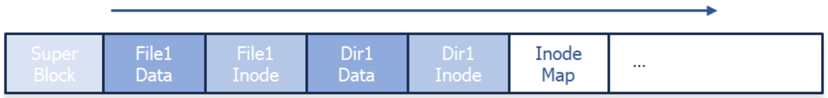
1. Introduction

Log-structured File System

FFS



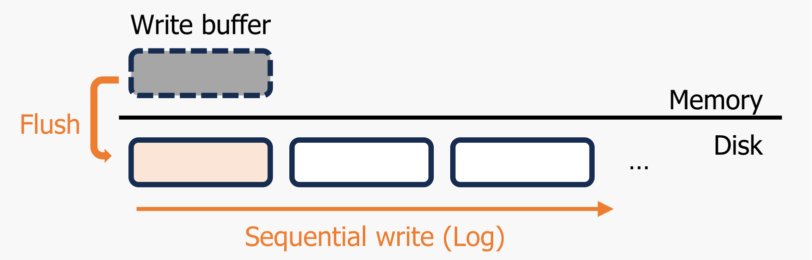
LFS



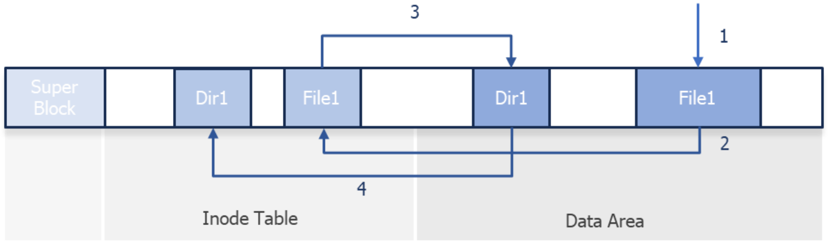
- Write all changes to disk sequentially in a single operation ➔ **Log**
- Convert small synchronous random writes to large asynchronous sequential transfers ➔ Write buffer

1. Introduction

Log-structured File System



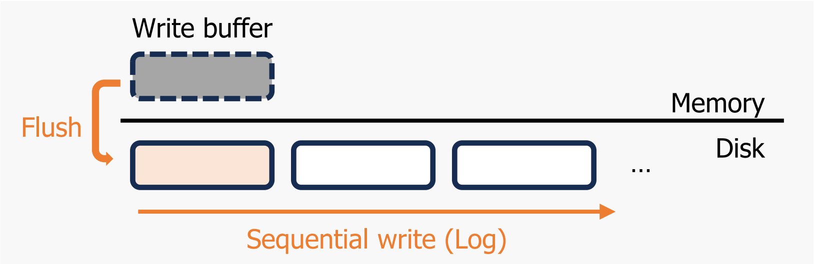
FFS



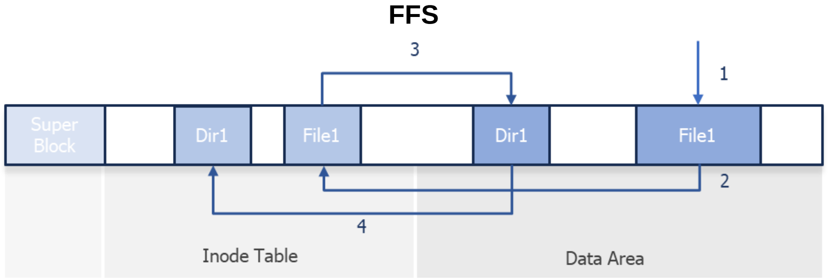
LFS

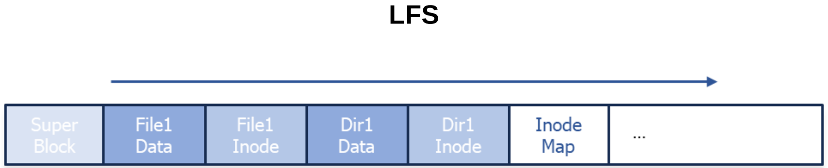
- Write all changes to disk sequentially in a single operation ➔ **Log**
- Convert small synchronous random writes to large asynchronous sequential transfers ➔ Write buffer

1. Introduction



Log-structured File System





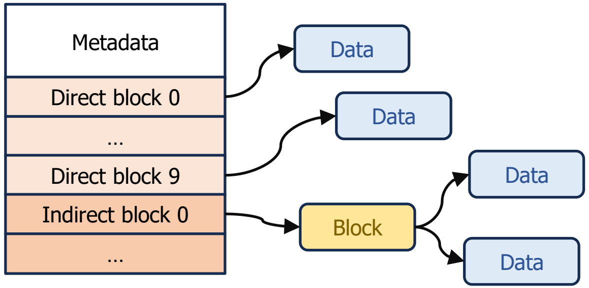
- Write all changes to disk sequentially in a single operation ➔ **Log**
- Convert small synchronous random writes to large asynchronous sequential transfers ➔ write buffer

1. How to retrieve information from log?

2. How to manage free space on disk?

2. Design

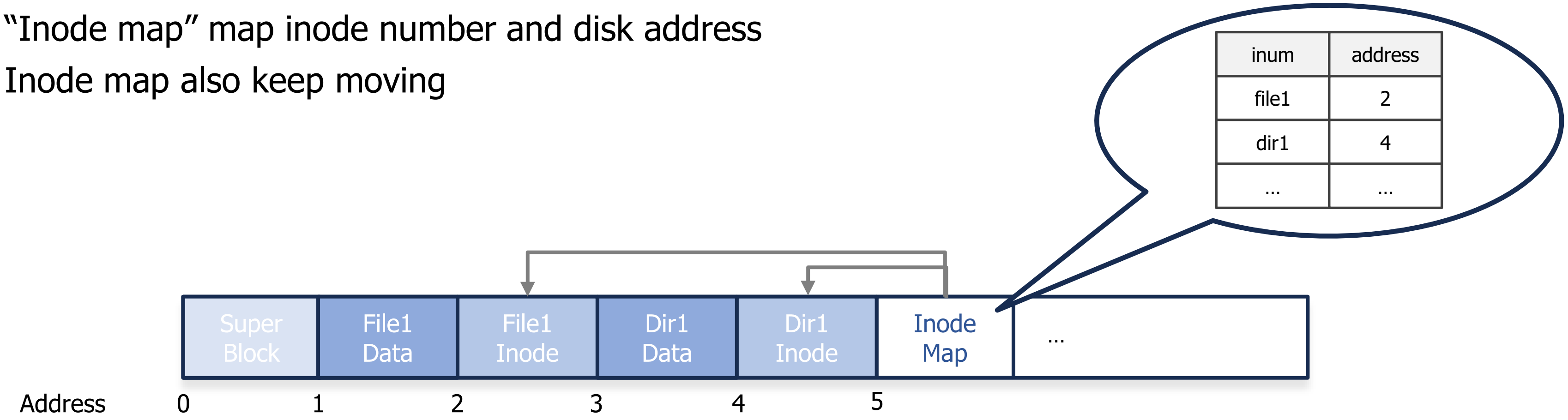
How to retrieve information from log?



Inode

Inode structure

- Same as FFS
- Written in append like data: not fixed position
- “Inode map” map inode number and disk address
- Inode map also keep moving

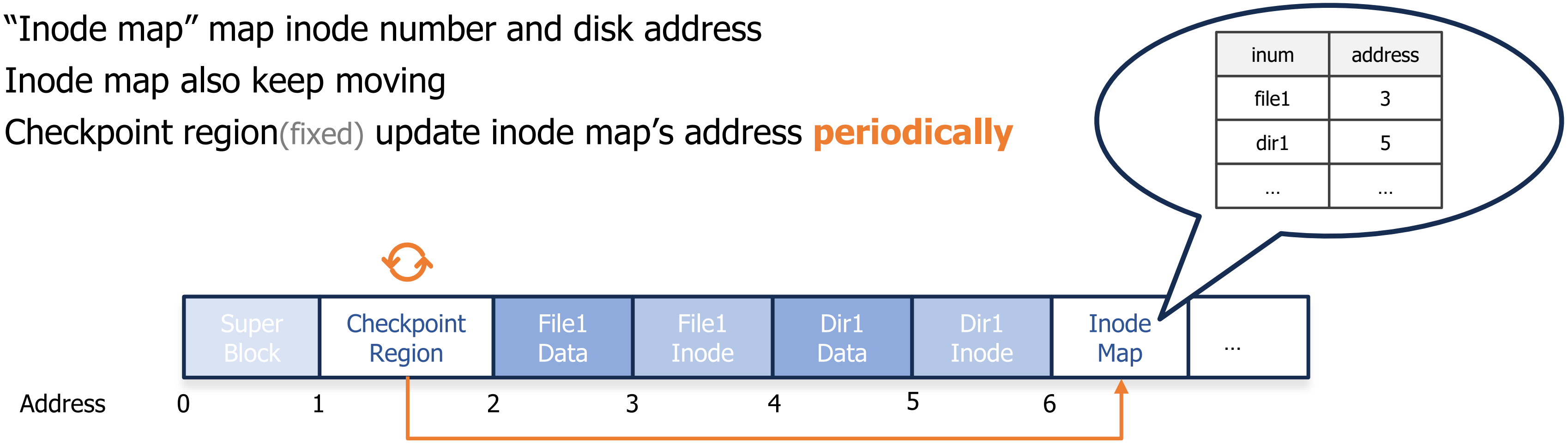


2. Design

How to retrieve information from log?

Inode structure

- Same as FFS
- Written in append like data: not fixed position
- “Inode map” map inode number and disk address
- Inode map also keep moving
- Checkpoint region(fixed) update inode map’s address **periodically**



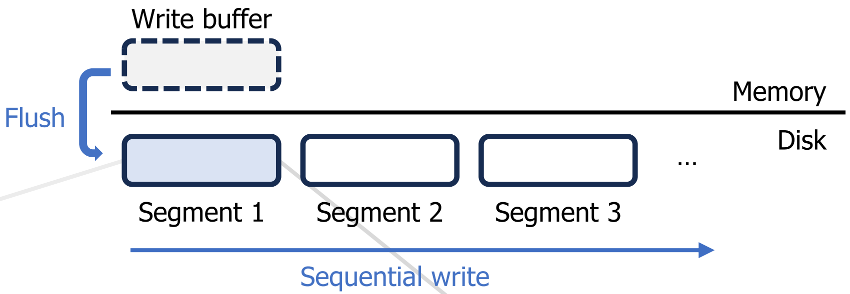
Segment summary block: identify contents of segment
Segment usage table: counts valid data and store last write time

2. Design

How to manage free space on disk?

Segment

- Large fixed-size extents (512KB ~ 1MB)
- Write in append
- Read data sequentially from beginning to end
  - Segment summary block contains the file number and block number (per write)
- Out-of-place update (need to be cleaned)



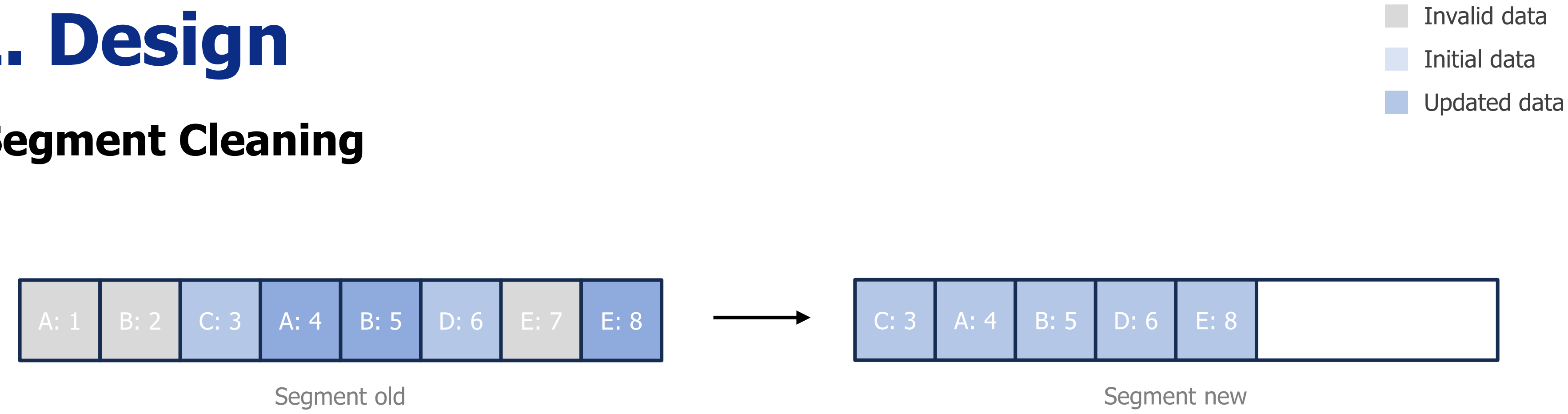
2. Design

Segment Cleaning

- Make free spaces by removing invalid data
- Executed when # of clean segments drops below threshold
- Clean a few tens of segments (typically 50~100) at a time
- Identify which blocks of each segment are valid (which block belongs to each file)
    - If block pointer is valid, it is a live (no free block list or bitmap)
        - It distinguishes blocks overwritten or deleted
    - It is also useful during crash recovery

2. Design

Segment Cleaning

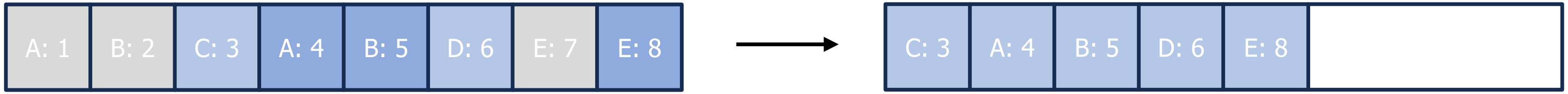


1. Read selected segments into memory
2. Identify valid data
3. Write valid data back to new segments

2. Design

Invalid data
Initial data
Updated data

Segment Cleaning



Segment old

Segment new

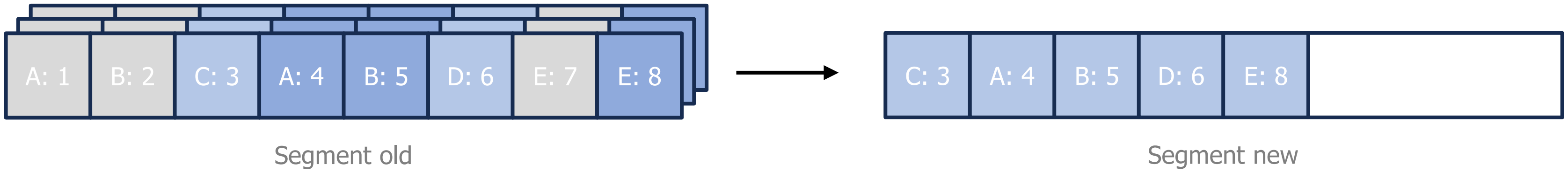
1. Read selected segments into memory
2. Identify valid data
3. Write valid data back to new segments

1. Which segmentation should be cleaned?
2. How should valid blocks be grouped when written out?

2. Design

Invalid data
Initial data
Updated data

Segment Cleaning



1. Read selected segments into memory
2. Identify valid data
3. Write valid data back to new segments

1. Which segmentation should be cleaned?
2. How should valid blocks be grouped when written out?

2. Design

Simulator

Settings

- Use 4KB files
- Overwrite one of files with new data
- Constant disk utilization(75%), no read traffic
- Run until all clean segments are exhausted

Random access pattern

- Uniform: same selection probability per segments
- Hot-and-cold: locality
  - Hot: 10% of files, 90% of times
  - Cold: 90% of files, 10% of times

2. Design

u: utilization of segments, still valid, inverse of bandwidth
N: count of segments

Segment Cleaning Cost

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

$$= \frac{\text{read} + \text{old valid write} + \text{new write}}{\text{new data written}}$$

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

- It includes all cleaning overheads
  (seek time, rotation latency, transfer time …)

- Write cost 1.0 is the best (no read, full bandwidth)

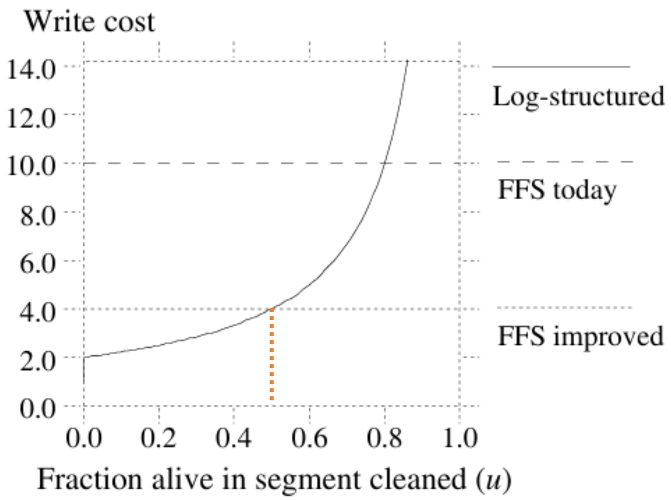


Figure 3 — Write cost as a function of u for small files.

2. Design

u: utilization of segments, still valid, inverse of bandwidth
N: count of segments

Segment Cleaning Cost

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

$$= \frac{\text{read + old valid write + new write}}{\text{new data written}}$$

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

- It includes all cleaning overheads
  (seek time, rotation latency, transfer time …)

- Write cost 1.0 is the best (no read, full bandwidth)

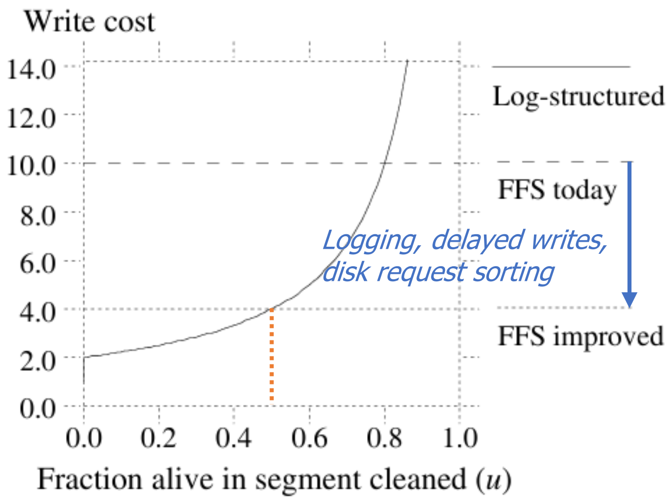


Figure 3 — Write cost as a function of u for small files.

2. Design

Segment Cleaning Cost based on Access Pattern

- Cleaner used greedy policy (least-utiilized segments)
- Segment utilization and disk utilization ⬇
  ➔ performance and storage cost⬆
- For high performance + low cost
  ➔ **bimodal segment distribution**
- Localized pattern(better grouping) result worse performance than no locality

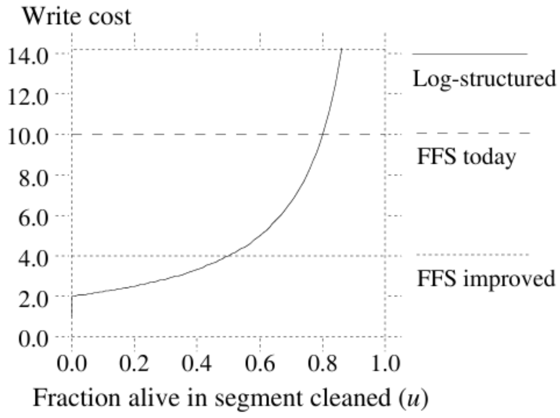


Figure 3 — Write cost as a function of *u* for small files.

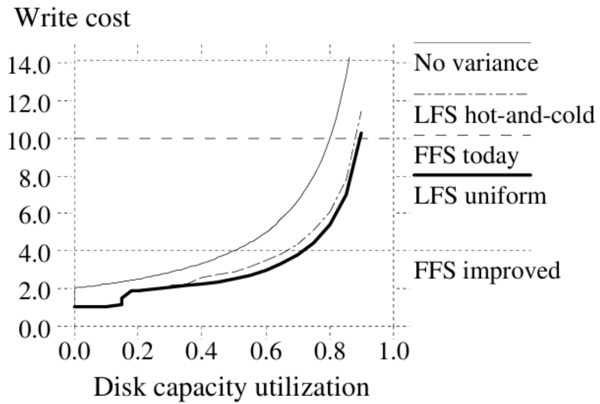


Figure 4 — Initial simulation results.

2. Design

Hot segments

Cold segments

Segment Selection Policy

Greedy

- Select least-utilized segments
  - Cold segments linger very long time (more valuable)
  - Hot and cold segments must be treated differently

Cost-benefit

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

- Select grouped segments by age
  - Allow cold segments to be cleaned at much higher than hot

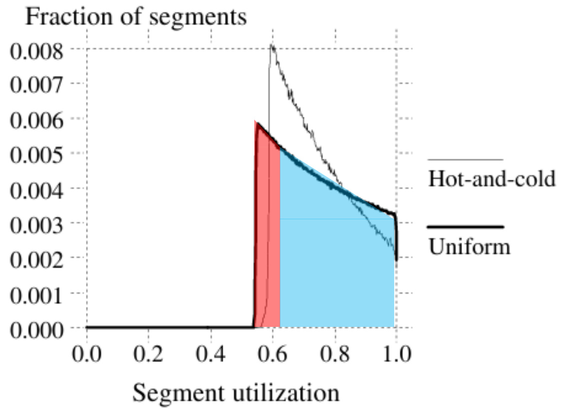


Figure 5 — Segment utilization distributions with greedy cleaner.

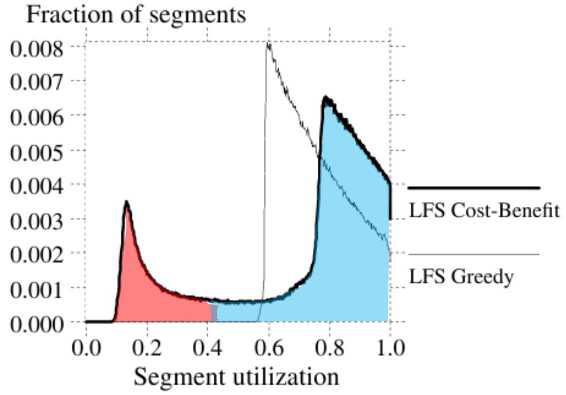


Figure 6 — Segment utilization distribution with cost-benefit policy.

2. Design

Cost-Benefit Policy

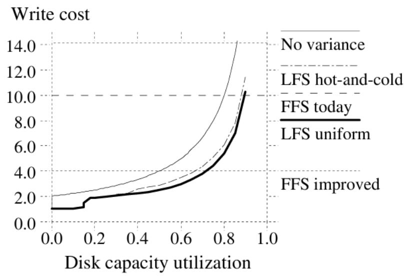


Figure 4 — Initial simulation results.

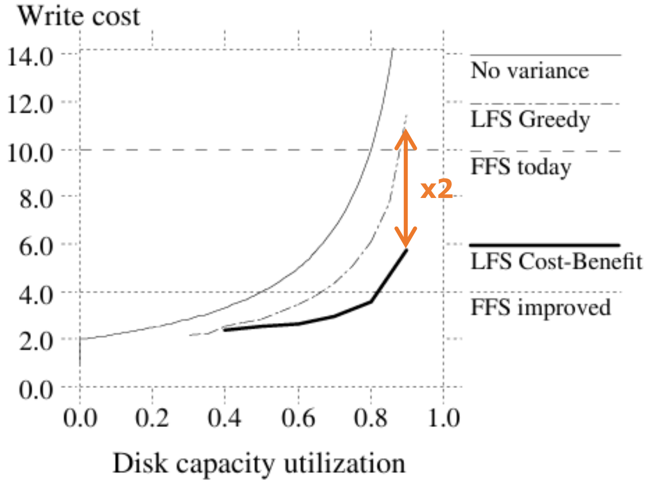


Figure 7 — Write cost, including cost-benefit policy.

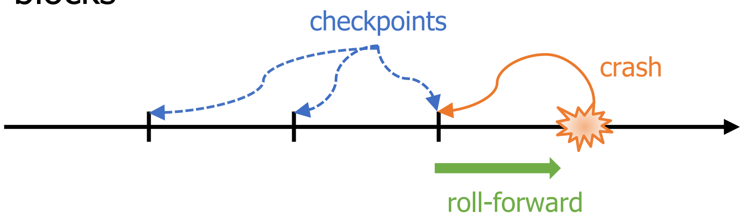
- Use cost-benefit policy in hot-and-cold access pattern
- Reduce write cost by as 50% over
- Get better even as locality increases

2. Design

Segment summary block: identify contents of segment
Segment usage table: counts valid data and store last write time

Crash Recovery

- If crashed, check the latest checkpoint: **roll-forward**
  - Because updating checkpoint region executed periodically, there is likely to be non-updated data
- Recover recently-written file data from segment summary blocks
  - If block indicates presence of new inode, update inode map
  - If not, new version of file on disk incomplete (ignore)



- Adjust utilizations in segment usage table
- To restore consistency between directories and inodes, log includes "directory operation log"

3. Evaluation

Environment

Comparison

- Sprite LFS
- SunOS 4.0.3

Machine

- Sun-4/260 with 32MB RAM
- Sun SCSI3 HBA
- Wren IV disk (1.3MB/sec): 300MB formatted
  - SunOS: 8KB block size
  - Sprite LFS: 4KB block size, 1MB segment

3. Evaluation

Benchmark

Assume no cleaning happened

Small file performance

- High speed for creation and deletion
- Disk busy:
  - Sprite LFS: 17%
  - SunOS: 85%

Large file performance

- Higher write bandwidth (faster for random I/O)
- Low performance in case of reading file sequentially after written randomly

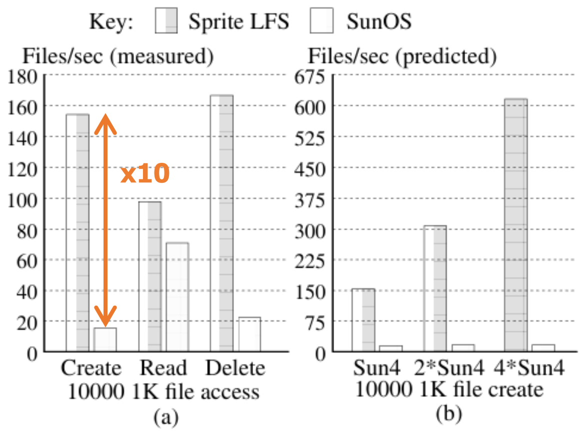


Figure 8 — Small-file performance under Sprite LFS and SunOS.

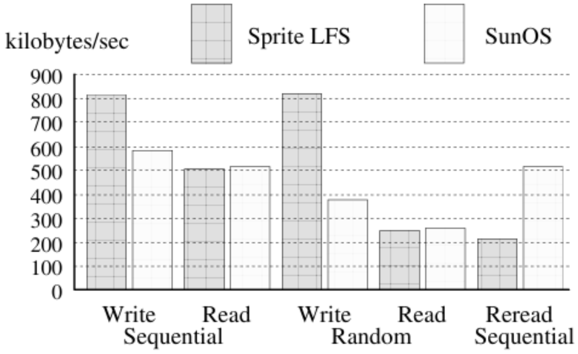


Figure 9 — Large-file performance under Sprite LFS and SunOS.

3. Evaluation

Record statistics over a period of 4 months

Cleaning Overhead with Real File System

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

Disk capacity utilization

Table 2 - Segment cleaning statistics and write costs for production file systems.

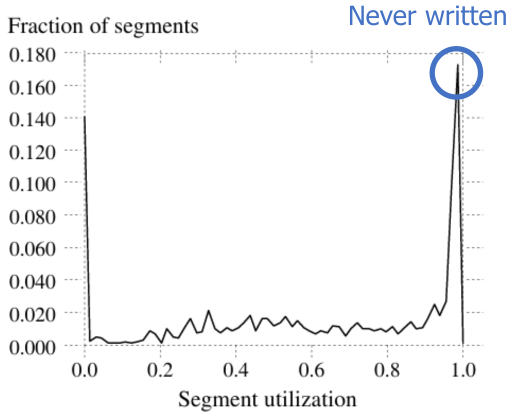


Figure 10 — Segment utilization in the /user6 file system

- Cleaning overhead limit long-term write performance to 70% of maximum sequential write bandwidth
- More than half of segments cleaned were totally empty
- There are a few differences between simulated and real-world environments (more locality)

4. Conclusion

- Disk random I/O performance is very bad, and existing FSs(like FFS) have many small synchronous random writes.
  We propose an LFS that uses a write buffering, append-only(log) approach to make all writes sequential, asynchronous, one large operation.

- To efficiently manage free space, we use a segment with a fixed size of large extent and perform a segment cleaning process to clean up invalid data.
  To determine which segments to clean, we adopt a cost-benefit policy that considers the age of the data.

- Compared to FFS, LFS outperforms in all cases except reads to large files.
  In practice environment, there is much overhead than predicted.

5. Key Take Away

	LFS	NAND flash based SSD
Structure	Append-only, Erase-Before-Write	
Write Unit	Extent (based on write buffer)	Page
Erase Unit	Segment	Block
Operation		
Free Space Mgmt.	Segment Cleaning	Garbage Collection
Mapping	Inode Map, Segment Summary Block	LPN to PPN

Q&A

Thank you!