The Design and Implementation of a Log-Structured File System

By
Christian Diercks
and
Rajarshi Chakraborty

Outline

- Introduction & History
- Existing File System Problems
- Log Structured File Systems
- File Location and Reading
- Free Space Management
- Segment Cleaning
- Simulation results for Sprite LFS
- Crash Recovery
- Experience with Sprite LFS
- Related Work
- Conclusion

Current tech trends

- CPU- disk gap increases
- No major improvements in transfer rate
- No major improvements in access time
- Exponential increase size of main memory

Technology Trends

Technology trends				
Technology		Year		
		1975	1985	1995
CPU	speed	1 MIPS	2 MIPS	1000 MIPS
Memory	speed	1 us	100 ns	1 ns
	capacity	.1 MB	10 MB	10000 MB
Disk	speed	60 ms	30 ms	8 ms
	capacity	100 MB	300 MB	3000 MB

Applications tend to become I/O bound!

Solution

Decouple disk bound applications from I/O

- Cache file data in main memory
- Decrease synchronous operations

Memory is cheap

- Increasing the cache helps for read type operations
- But what happens with the write performance?

Write operations

- Data needs to be flushed out to disk for safety reasons
  - Disk performance becomes dominated by write operations
  - Executing writes as soon as they occur reduces traffic, but less than 5% of the potential bandwidth is used for new data

The rest of time is spent seeking

Existing File System Problems

Two General problems:

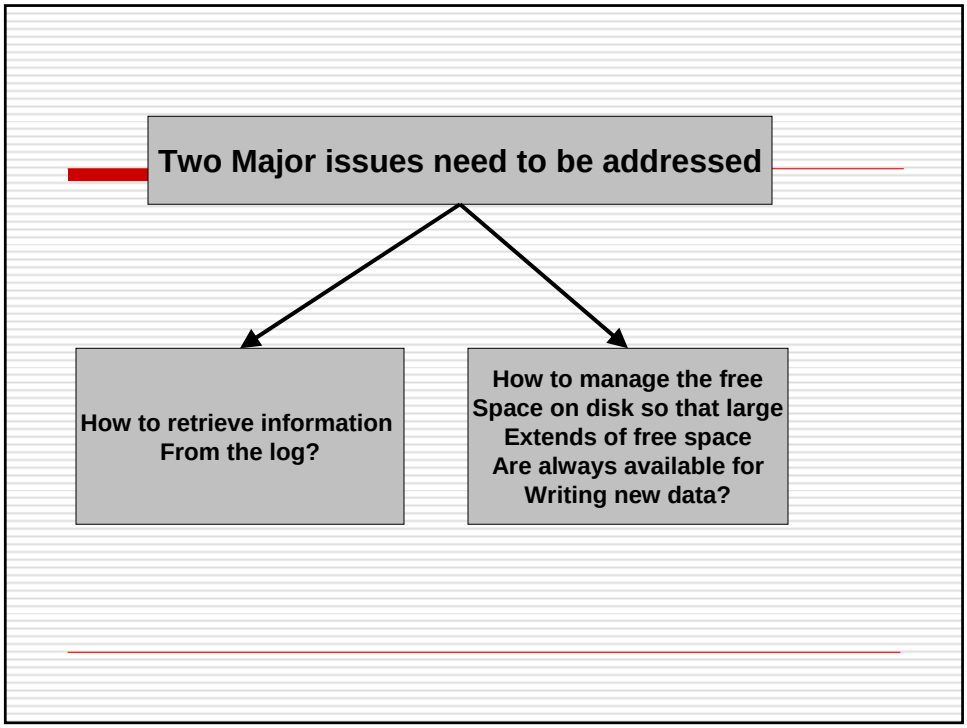
1. Data is spread around the disk in a way that causes too many small accesses
2. Write Synchronously - The application has to wait for a write to complete
   - Metadata is written synchronously
   - Small file workload make synchronously metadata writes dominating

Key Points of LFS

- Use file cache to buffer a sequence of file system changes
- Write the changes to disk sequentially in a single disk write operation

Log-Structured File Systems

- Small file performance can be improved
  - Just write everything together to the disk sequentially in a single disk write operation
- Log structured file system converts many small synchronous random writes into large asynchronous sequential transfers.
  - Nearly 100% of raw disk BW can be utilized

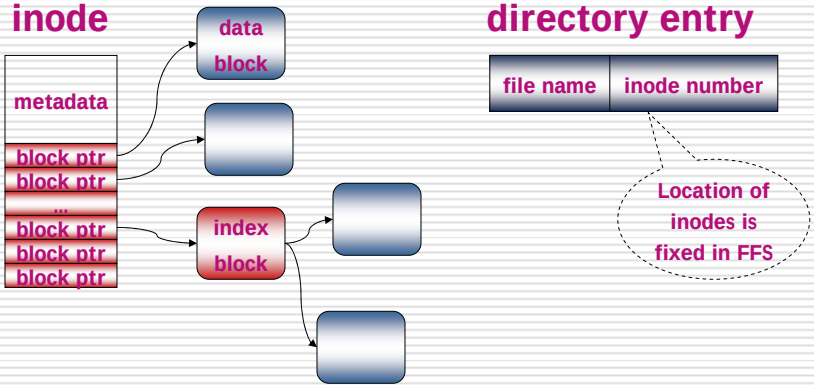


On Disk Data Structures

Data Structure	Location
I-node	Log
I-node Map	Log
Indirect block	Log
Segment summary	Log
Segment usage table	Log
Superblock	Fixed
Checkpoint region	Fixed
Directory change log	Log

Logical structure of file

- Indexed structure is the same as FFS

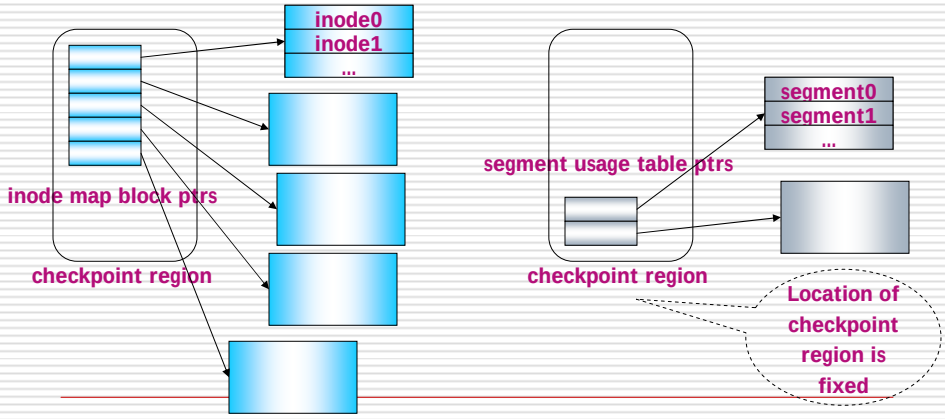


How to locate and read a File?

- Designer Goal
  - Match read performance of Unix FFS by using index structures in the log to allow random access
- I-node maps
  - Used to find disk location of Inode
  - Kept in cache to avoid many disk accesses
  - Maps are divided into blocks that are written to segments on disk (Checkpoint)

Inode Map and Segment usage Tab.

- Inode map
- Segment usage table



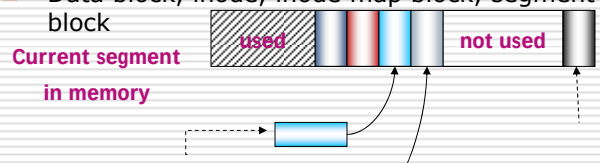
Read and Write Operation

- Read a block
  - Inode map block ptr -> inode map block -> inode -> data block

In memory

Same as FFS

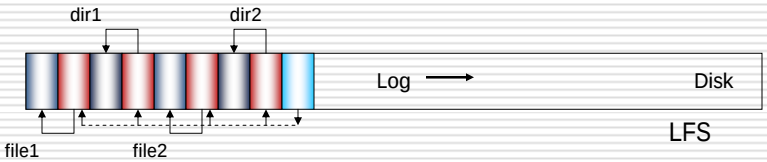
- Write a block
  - Data block, inode, inode map block, segment usage table block



  - Update inode map table ptr, segment summary block, segment usage table

Layout of disk LFS vs. FFS

- Example of creating 2 files in different directories



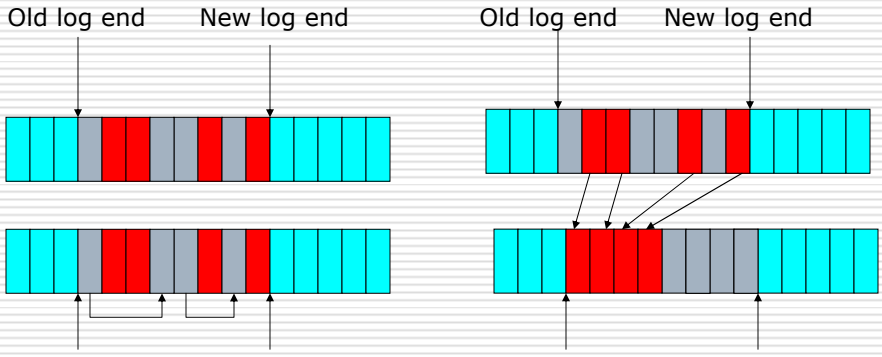
Inode | Directory | Data | Inode map

Location of inodes is not fixed in LFS

Free Space Management

- What to do when log wraps around on disk?
  - By this time earlier data is likely quite fragmented.
  - Possibly data "earlier" on the log is not valid at this point (not "live"!) and can be discarded.
  - By consolidating such data, the LFS is able to continue logging.
- From this point on two solutions to this problem:
  - Threading: do not touch "live" data – copy new data on the areas of the log that are "dead" at this point.
  - Potential problem: excessive fragmentation.
  - Plus: easy to conceptualize and do (just thread all live areas together).
  - Copy Live Data: bring live data together in the log
  - This implies that some Garbage Collection takes place
  - GC may become expensive (due to overheads).
  - Long-lived files may get copied many times.

Threading/Copy and Compact



Threaded Log

Copy and Compact

Segmented Logs in LFS

- LFS uses both threading and copying
  - Keep write long/large
- Disk divided into large fixed-size extents called ***segments.***
  - Any given segment always written sequentially start to end.
    - Live data must be copied out before its rewritten
  - Log is threaded segment by segment.

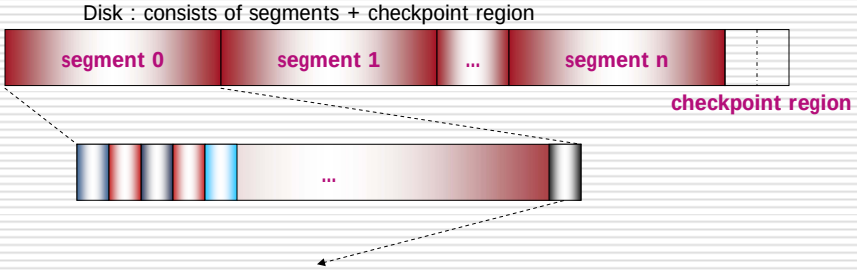
- Segments are fixed and large size extents:
  - Segment sizes are chosen large enough so that overhead for seek is small.
  - They yield a high fraction of disk bandwidth (even with some seeks in between).

- Segment writes contain:
  - Data for multiple files.
  - I-node related information
  - Directory data and I-node changes
  - I-node map (where to find the I-nodes for the files).

Segments

- Segment : unit of writing and cleaning
  - 512KB ~ 1024KB

Disk : consists of segments + checkpoint region

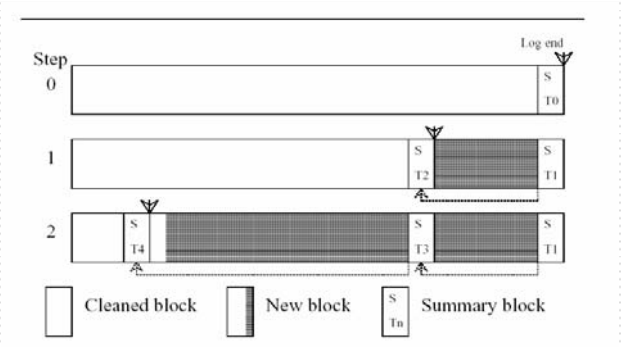


- Segment summary block
  - Contains each block's identity : <inode number, offset>
  - Used to check validness of each block

Segment Summary Block

- This block contains:
  - Each piece of information in the segment is identified (file number, offset, etc.)
  - Summary Block is written after every partial segment write
  - Helps in deciding the block's liveness
- There is no free-list or bitmap in LFS.

Multi Log Writing



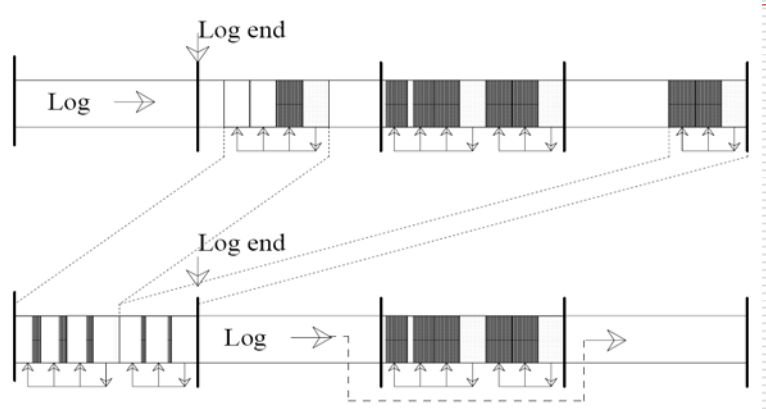
Cleaning of Segments

- Process of copying live data out of a segment is called cleaning.
- To free up segments, copy live data from several segments to a new one (ie, pack live data together).
  - Read a number of segments into memory
  - Identify live data
  - Write live data back to a smaller number of clean segments.
  - Mark read segments as clean.
- Various issues:
  - How to identify "live" objects in a segment?
  - How to identify file and offset of each live block?
  - How to update that file's I-node with new location of live blocks?

Cleaning of Segments

- Segment Cleaning Questions:
- When to clean?
- How many segments to clean?
- ***Which segment to clean?***
  - Most fragmented?
- ***How should live data be sorted when written out?***
  - Enhance locality of future reads?
  - Age sort?
  - Sprite LFS starts cleaning segments when the number of clean segments falls below a threshold (a few tens).
  - Cleans a few tens at a time until the number of clean segments surpasses another threshold.

Cleaning of Segments cont.



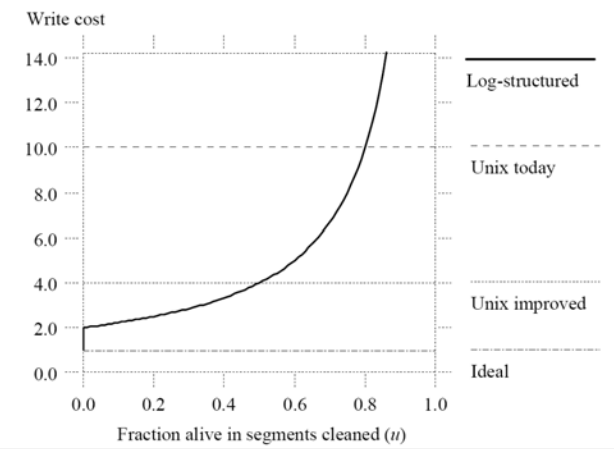
Write Cost

- *Write cost*: measure how busy the disk is per byte of new data written
  - Used to compare cleaning policies.
  - Includes segment cleaning overhead
  - Ignore rotational latency – look just at number of bytes
  - 1.00 is perfect – no cleaning overhead
  - 10 means that 1/10 of disk time is spent writing new data
- Write cost is ***estimated*** as:
  - (total bytes read and written)/(new data written)
  - (read segs + write segs + write new) / (new data written)
- If average utilization of live data in segments is u:
  - Read N segments
  - Write N*u old data
  - Leaves space N*(1-u) of new data
  - Assumes segment must be read in entirety to recover live data. If (u==0) then no need to read segment and the write cost is 1.

Write Cost as Function of u

- Need utilization below .8 or .5
  - NOT overall disk utilization; rather, fraction of live blocks in segments that need to be cleaned.
  - Performance can be improved by reducing disk utilization:
    - Less of disk in use : lower write cost
    - More of disk in use: higher write cost.
- Best: Bimodal distribution. Mostly full segments; work with cleaning mostly empty ones.

Write Cost as Function of u



Simulation Results

- Analyze different cleaning policies under controlled conditions
- FS – Fixed number of 4-KB files
- Every step overwrites a file with new data

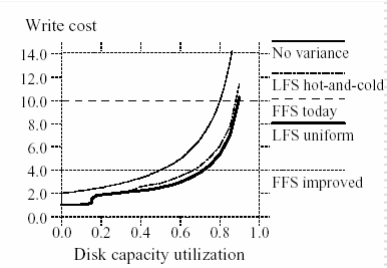
Access patterns to overwrite files:

- Uniform : Equal probability of being selected
- Hot-and-cold : 10% "hot" files selected 90% of the time, 90% "cold" files selected 10% of the time; HOT=>short-lived data & COLD=>long-lived data
- Equal chance of selection within each group

Simulation Results (Contd.)

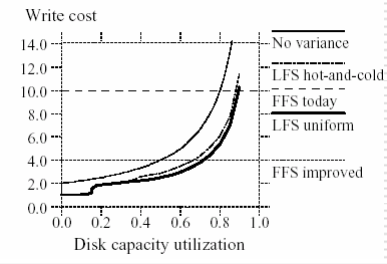
- Fixed overall disk capacity utilization
- No read traffic – only "write" for simulation
- Exhaust clean segments → Clean segments till a *threshold*
- Experiments run till the write cost stabilized

Initial simulation results



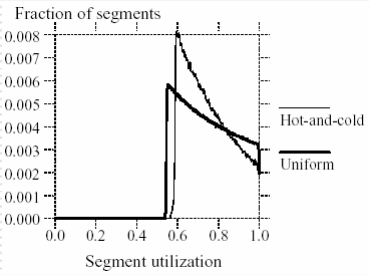
- Greedy policies for both *Uniform* and *Hot-and-Cold*
- *Uniform* – least-utilized segments cleaned
- *Uniform* – no sorting of live data in the selected segment
- *Hot-and-Cold* – sorts live data by age in the selected segment

Initial simulation results



- Locality & better grouping give worse performance than no-locality system
- *Hot-and-cold* with 95%-5% => worse performance

Segment utilization with greedy cleaner



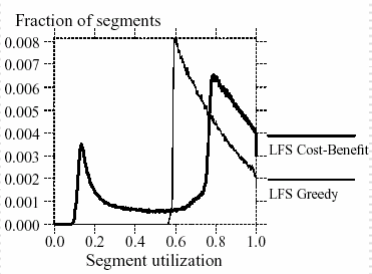
- Only least utilized of all segments is cleaned
- Utilization of every segment drops to the cleaning threshold
- Drop in utilization of the cold segments is slow
- More segments clustered around the *cleaning point* for "locality" than in "non-locality" based simulation

Cost-Benefit Policy

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

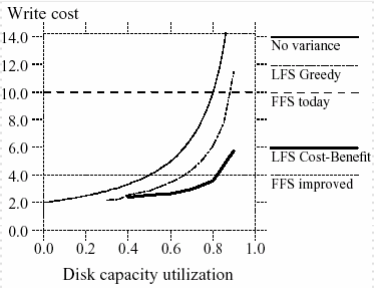
- The cleaner now chooses the segment with the highest *benefit-to-cost* ratio to clean
- Benefit: (1)free space reclaimed (2)amt. of time to stay free
- u = utilization of segment
  - Free space: 1-u
  - Time: Age of the youngest block
  - Cost = 1 read cost + u writing cost (1+u)

Cost-benefit vs Greedy



- Bimodal: cleans *cold* at 75% (u) and *hot* at 15% (u)
- 90% of writes for *hot* means mostly *hot* segments are cleaned

Why choose cost-benefit policy?



- 50% less write cost than *greedy policy*
- Even outperforms the best possible Unix FFS at high disk capacity utilization
- Gets better with increasing locality

Segment usage table

Number of live bytes in the segment	Age of the youngest block in the segment

Segment-usage table record for each segment

- Supports cost-benefit
- Values are set when the segment is written
- Segment reused only when # of live bytes = 0
- Blocks of this table kept in the *checkpoint regions* (which is used in crash-recovery)

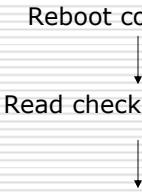
Crash recovery

- Traditional Unix FS must scan ALL metadata for restoring consistency after reboot – takes a lot of time with increasing storage size
- LFS – last operation at the end of the log (also used in other FS and databases) – quicker crash recovery
  1. *Checkpoint*
  2. *Roll-forward*

Checkpoints

- Position in the log – FS is consistent & complete
- 2-phase process to create a checkpoint
  1. Write all modified info to the log
  2. *Checkpoint region* – fixed position on disk; checkpoint region -> all blocks in inode map & segment usage table, current time & last segment written
  3. 2 checkpoint regions

Checkpoints (Contd.)



- Two checkpoint regions to handle crash during checkpoint operations
- Checkpoint time is in the last block of the region
- System uses the most recent time; time for the failed checkpoint is not recorded

Creation of a checkpoint

1. Periodic intervals
2. File system is unmounted
3. System is shutdown

Controlling recovery time (Contd.)

- Longer interval => less checkpoint overhead, more recovery time
- Shorter interval => less recovery time, costlier normal ops
- Alternative: Checkpoint after a certain amount of new data written

Roll-forward

- Scanning BEYOND the last checkpoint to recover max. data
- Use information from *segment summary blocks* for recovery
- If found new inode in Segment Summary block -> update the inode map (read from checkpoint) -> new data block on the FS
- Data blocks without new copy of inode => incomplete version on disk => ignored by FS

Roll-forward (Contd.)

- Adjusting utilization in the segment usage table to incorporate live data after roll-forward (utilization after checkpoint = 0 initially)
- Adjusting utilization of deleted & overwritten segments
- Restoring consistency between directory entries & inodes

Restoring consistency between directories and inodes

1. Special log record for each directory change – *Directory operation log*

   <operation code, location of dir. entry, contents, new ref. count for inode in the entry>
2. Log entry exists but no inode/directory block => update and append the directories, etc to the log and create a *new checkpoint*
3. Checkpoint represents *consistency* between directory operation log and inodes /directory blocks in the log

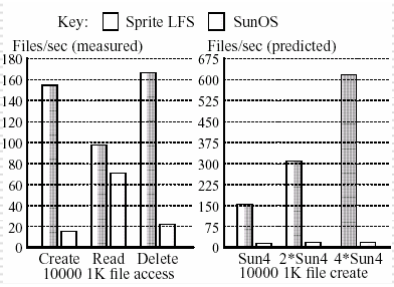
Implementation of Sprite LFS

- Began in late-1989 and operational by mid-1990!
- Implemented for the Sprite Network OS – installed in 5 partitions and used by 30 users
- Roll-forward not implemented yet ☹
- Short checkpoint interval – 30 sec
- Data after last checkpoint discarded after reboot

Sprite LFS vs. Unix FFS

Sprite LFS	Unix FFS
Additional complexity for implementing segment cleaner	No segment cleaner implemented
No bitmap / layout policies required implemented	Bitmap / layout policies makes it as complex as segment cleaner
Recovery code	*fsck* code

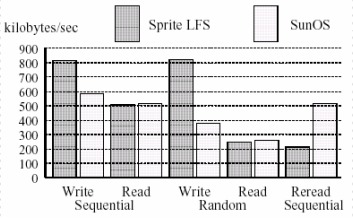
Micro-benchmarks: small files



- Best-case performance – *no cleaning*
- Sprite LFS vs. SunOS 4.0.3 (based on Unix FFS)
- Sprite LFS: segment size = 1MB, block size = 4 KB
  SunOS: block size = 8KB

- Sprite kept disk 17% busy while saturating CPU; SunOS saturated disk 85% - only 1.2% of potential disk bandwidth used for new data – Sprite WINS!

Micro-benchmarks: large files



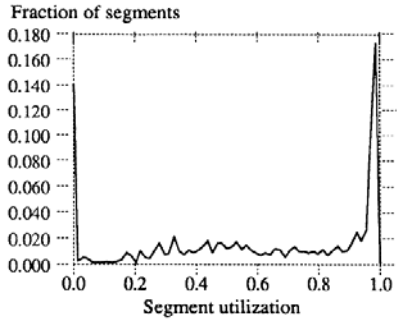
- Sequential rereading requires seeks in Sprite, hence its performance is lower than SunOS
- Traditional FS – *logical locality* (assumed access pattern)

  Log-structured FS – *temporal locality* (group recent created/modified data)

Cleaning overheads

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



- Collected over a 4-month period
- Better performance than predicted through simulation – low write cost range
- Segment utilization of /user6 partition
- Large number of fully utilized and totally empty segments

Crash recovery time

Sprite LFS recovery time in seconds			
File Size	File Data Recovered		
	1 MB	10 MB	50 MB
1 KB	1	21	132
10 KB	< 1	3	17
100 KB	< 1	1	8

- Code can time recovery of various crash scenarios
- Less data written between checkpoints => less recovery time
- Also dependent on number of files written between checkpoints

Related Work

- Previously implemented on write-once media – no reclaiming of log
- Segment cleaning Ξ garbage collection in programming languages
- One block <-> one file: garbage identifying algo. is simpler
- Database use write-ahead logging for crash recovery
- Similar to *group commit* in database systems

Conclusion

- The motivation for designing a new log-structured file system
- Design and architecture of the Sprite LFS – including issues like *free space management* and *segment cleaning*
- Simulation-based study to choose the right design for implementation
- Implementation-based study and comparison to prove the superiority of Sprite LFS over traditional FFS

Reference

- Rosenblum, M. and Ousterhout, J. K. "The Design and Implementation of a Log-Structured File System" ACM transaction on Computer Systems, Vol 10, No. 1, February 1992, pp. 26-52
- Rosenblum, M. "The Design and Implementation of a Log-Structured File System"

Dhanyabad!

Thank You!

Danke!