The Design and Implementation of a Log-Structured File System

CS380L: Emmett Witchel

Treat disks like tape.

J. Ousterhout

File system design is governed by two general forces: technology, which provides a set of basic building blocks, and workload, which determines a set of operations that must be carried out efficiently.

Rosenblum and Ousterhout

1 Preliminaries

1.1 Outline

- Introduction
- 2 hard problems
  - Finding data in log
  - Cleaning
- 2 key ideas
  - logging and transactions: log your writes
  - indexing: inode $\rightarrow$ data can live anywhere $\rightarrow$ no need to "write back"

2 Introduction

2.1 Why this is "good" research

- Driven by keen awareness of technology trend
- Willing to radically depart from conventional practice
- Yet keep sufficient compatibility to keep things simple and limit grunge work
- 3-level analysis:

- Provide insight with simplified math – “science”
- Simulation to evaluate and validate ideas that are beyond math
- Solid real implementation and measurements/experience

- Extreme research – take idea to logical conclusion (e.g., optimize file system for writes since “reads will all come from the cache”)

2.2 Technology trends

- Memory will grow, all reads will come from disk cache
  - Is this true today? Why or why not?
- Transfer bandwidth and access time. Where has this gone?
- RAIDs and network RAIDs
- File system design is governed by two general forces: technology, which provides a set of basic building blocks, and workload, which determines a set of operations that must be carried out efficiently.

2.3 Implications

- Reads taken care of (?)
- Writes not, because paranoid of failure
- Most disk traffic is writes
- Can’t afford small writes
  - RAID5 makes small writes worse
- Simplify and make FS less “device-aware”
  - No tracks, cylinders, etc
  - Just “big writes fast” + temporal locality between write and read patterns

2.4 Problems with UNIX FFS

- (Because most files are small)
- Too many small writes
- (Because of recovery concerns)
- Too many synchronous writes

- It takes at least five separate disk 1/0s, each preceded by a seek, to create a new file in Unix FFS: two different accesses to the file's attributes plus one access each for the file's data, the directory's data, and the directory's attributes. When writing small files in such a system, less than 5% of the disk's potential bandwidth is used for new data; the rest of the time is spent seeking.

2.5 Approaches

- Replace synchronous writes with asynchronous ones (dribble latest updates to disk)
- Replace many small writes with a few large ones
- So buffer in memory and write to disk using large "segment-sized" chunks
- Log-append only, no overwrite in place

2.6 Key difference between LFS and other log-based systems:

- The log is the only and entire truth, there's nothing else

2.7 Challenges

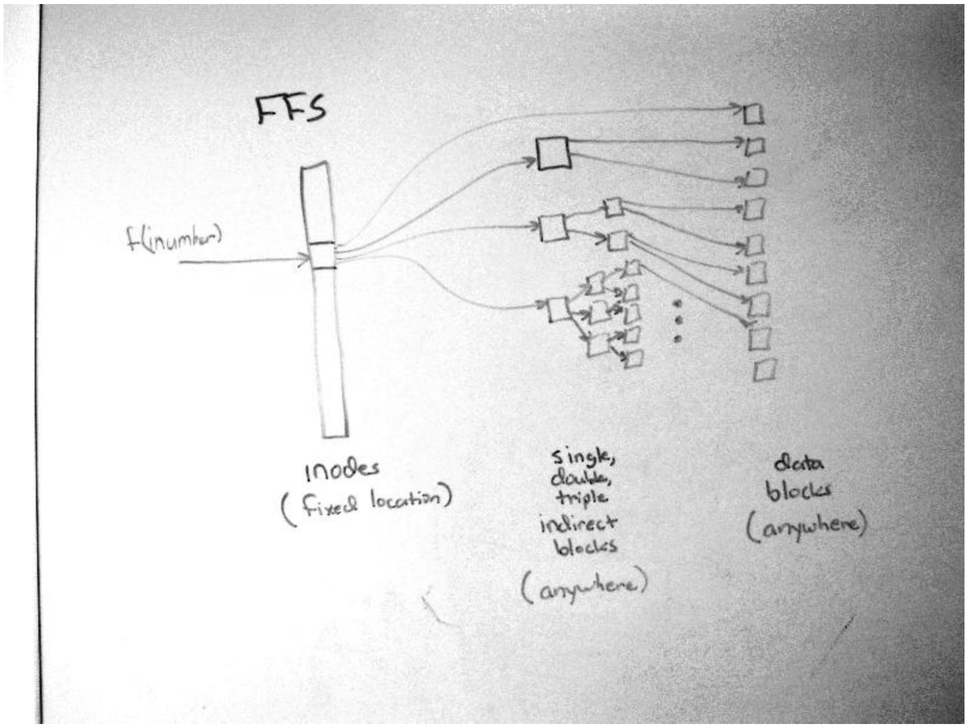
- Two hard problems
  - Metadata design
  - Free space management
- No update-in-place,
  - (almost) nothing has a permanent home,
  - So how do we find things?
- Free space gets fragmented,
  - So how to ensure large extents of free space?

2.8 The poetry of LFS

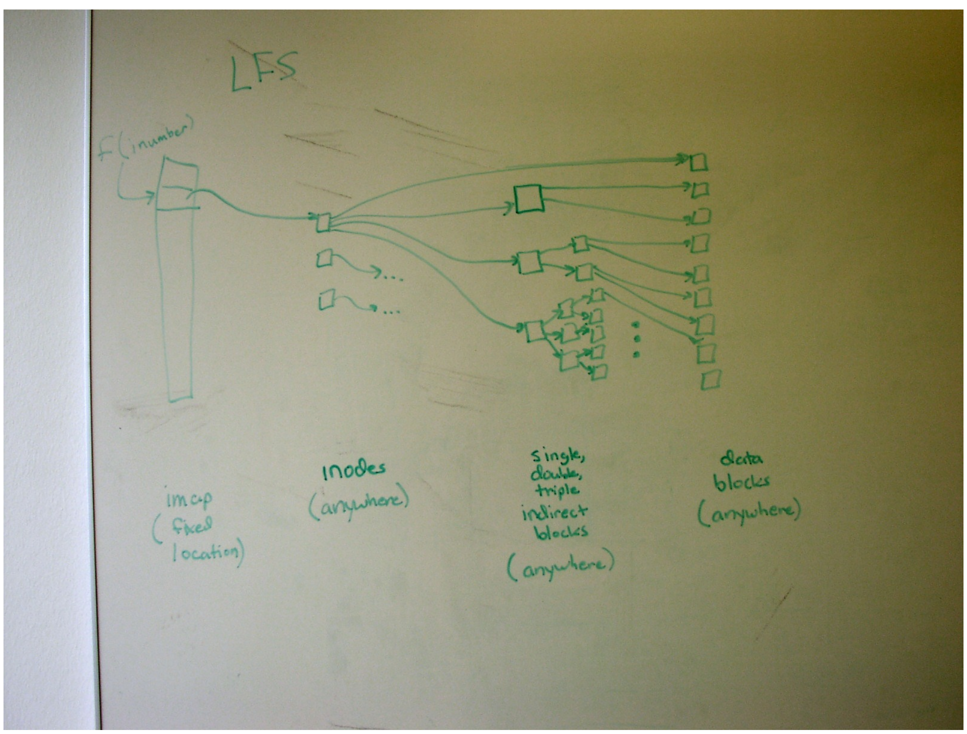
- Overall, Sprite LFS permits about 65-75% of a disk's raw bandwidth to be used for writing new data (the rest is used for cleaning).
- For comparison, Unix systems can only utilize 5-10% of a disk's raw bandwidth for writing new data; the rest of the time is spent seeking.
- The fundamental idea of a log-structured file system is to improve write performance by buffering a sequence of file system changes in the file cache and then writing all the changes to disk sequentially in a single disk write operation.
- Basic operation, page 7

3 Index structures

3.1 Index structures in FFS



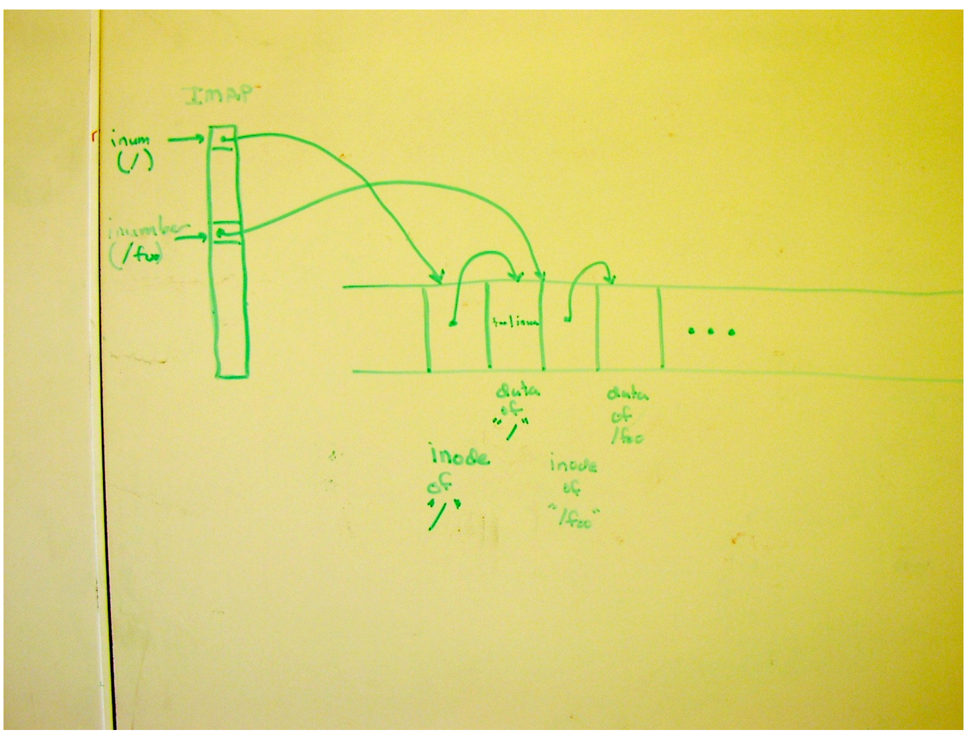
3.2 Index structures in LFS



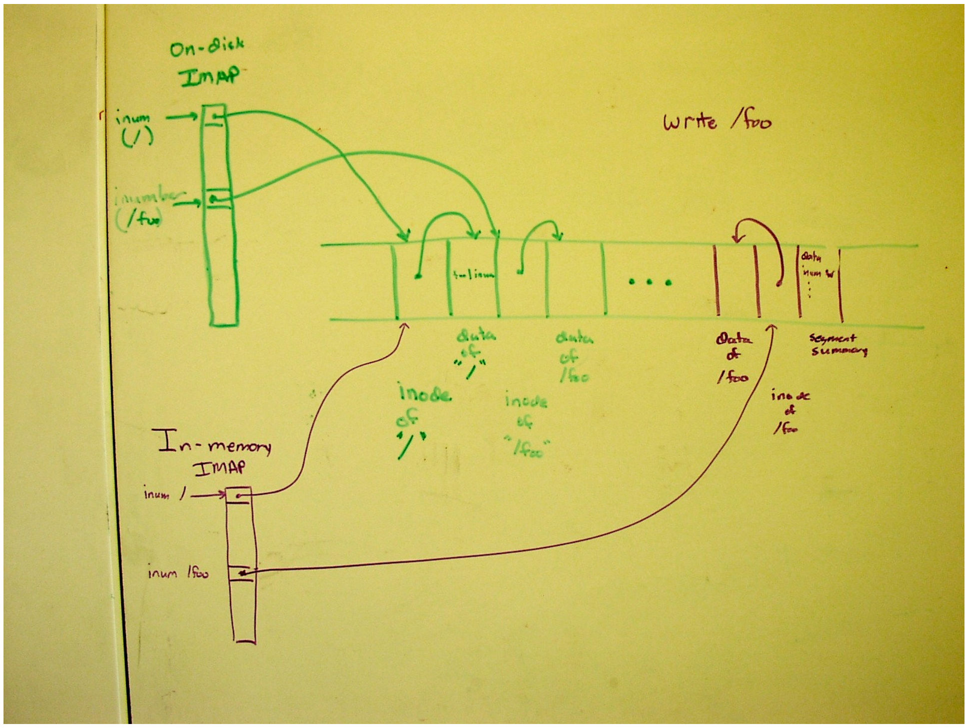
- Step 1 – move inodes to log
- Step 2 – find mobile inodes with fixed imap
- Not obvious this is better
  - Why is this better?
  - Don't have to write imap after every write – just at checkpoints; otherwise roll forward
  - Couldn't you do this with original inode array?
  - Would there be any advantages to making imap mobile by adding another level of indirection?
- Compare different checkpoint organizations: entire disk, inodes, imap, imap map, ...
  - Assume 100 bytes/inode, 4 bytes per disk pointer. 50 MB/s bandwidth.
  - Assume 512MB main memory

	disk data	inode array	imap	imap map
size	100 GB	1GB	40MB	320 KB
time to write checkpoint	2000 sec	20 sec	1 sec	10ms + seek + rot
Fraction of main memory	200x	2x	5%	.05%

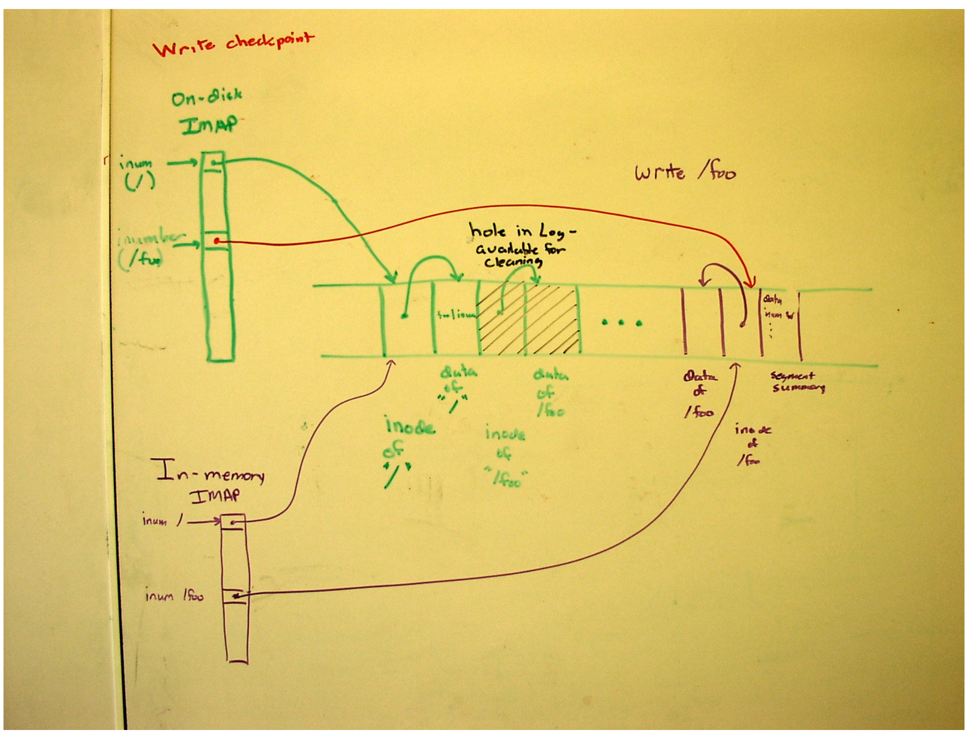
3.3 Example of LFS update



- Read "/foo"



- Write "/foo"
- Read "/foo" using in-memory imap
- What if we crash?



- Update checkpoint (eventually)

4 Cleaning

How to get back free disk space

4.1 Option 1: threading

- Put new blocks wherever holes are
- Each block written has points to next block in sequential log
- Advantage: Don't waste time reading/writing live data
- Law of storage system entropy: left to itself, free space gets fragmented to the point of approaching your minimum allocation size

4.2 Option 2: Compact the log

- Compact live blocks in log to smaller log
- Advantage: Creates large extents of free space
- Problem: Read/write same data over and over again

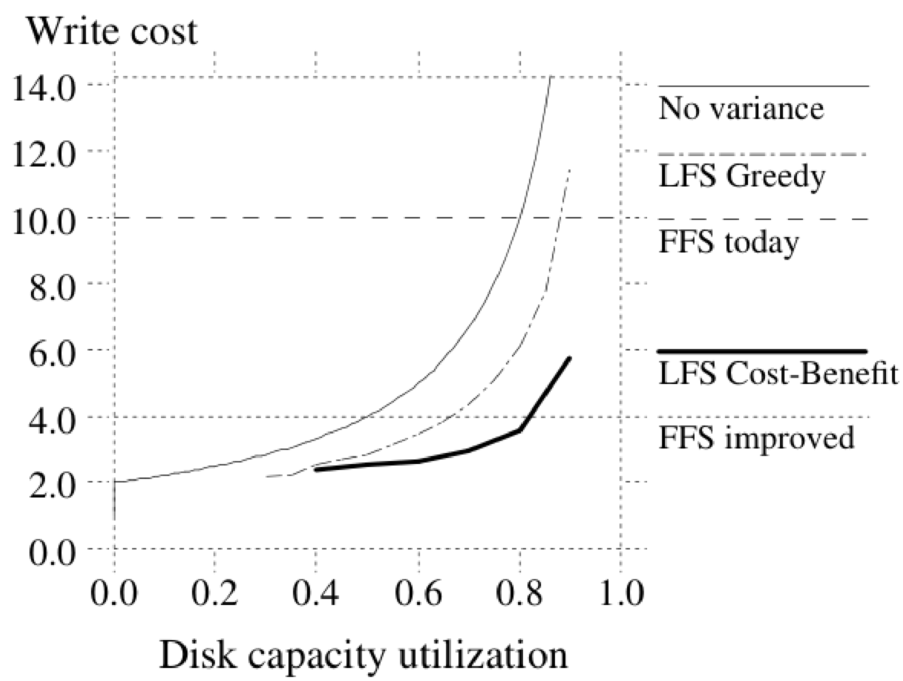
4.3 Option 3: Segments: Combine threading + compaction

- Want benefits of both:
  - Compaction: big free space
  - Threading: leave long living things in place so I don't copy them again and again
  - Easily detect dead blocks by having a version number with inode. If old version, no need to chase down inode pointers or indirect blocks.
  - The version number combined with the inode number form a unique identifier (uid) for the contents of the file.
  - if the uid of a block does not match the uid currently stored in the inode map when the segment is cleaned, the block can be discarded immediately without examining the file's inode.
- Solution: "segmented log"
  - Chop disk into a bunch of large segments
  - Compaction within segments
  - Threading among segments
  - Always write to the "current" "clean" segment, before moving onto next one
  - Segment cleaner: pick some segments and collect their live data together (compaction)

- Segment summary information
    * It must also be possible to identify the file to which each block belongs and the position of the block within the file; this information is needed in order to update the file's inode to point to the new location of the block.
    * Sprite LFS also uses the segment summary information to distinguish live blocks from those that have been overwritten or deleted.

- Many similarities with generational garbage collection

5 Policies, evaluation



5.1 Is cleaning going to hurt?

- Write cost = total_IO / new_writes = read segs + write live + write new / write new = [1+u+(1-u)]/(1-u) = 2/(1-u)
    - where u is utilization of segments cleaned

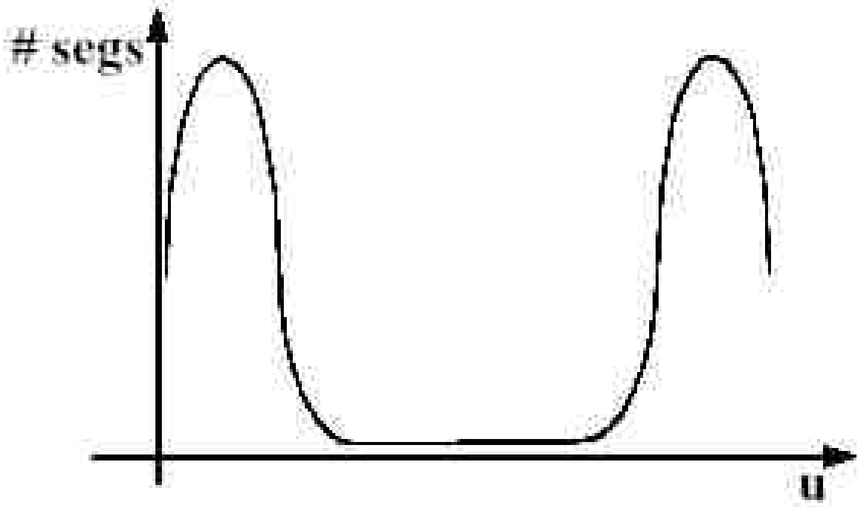
- A write cost of 10 means that only one-tenth of the disk's maximum bandwidth is actually used for writing new data; the rest of the disk time is spent in seeks, rotational latency, or cleaning.

- Conclusion: u better be small or it's going to hurt bad.

- Aha: u doesn't have to be overall utilization, just utilization of cleaned segments.

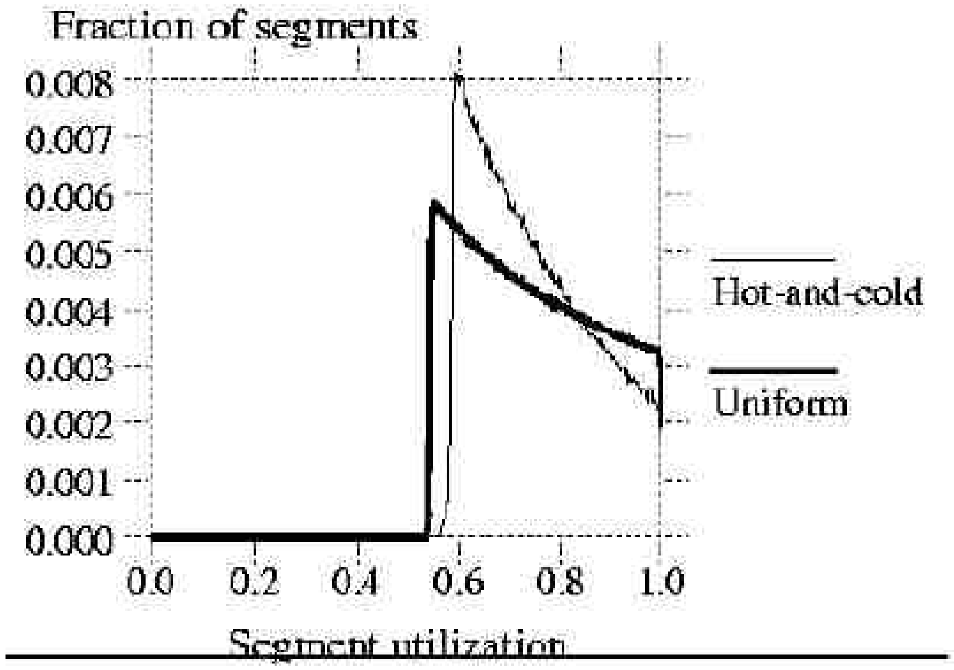
- FFS has a write cost of 10-20, corresponding to 5-10% disk bandwidth between seeks. (Figure 3, page 11)

5.2 How to lower cost under utilization?



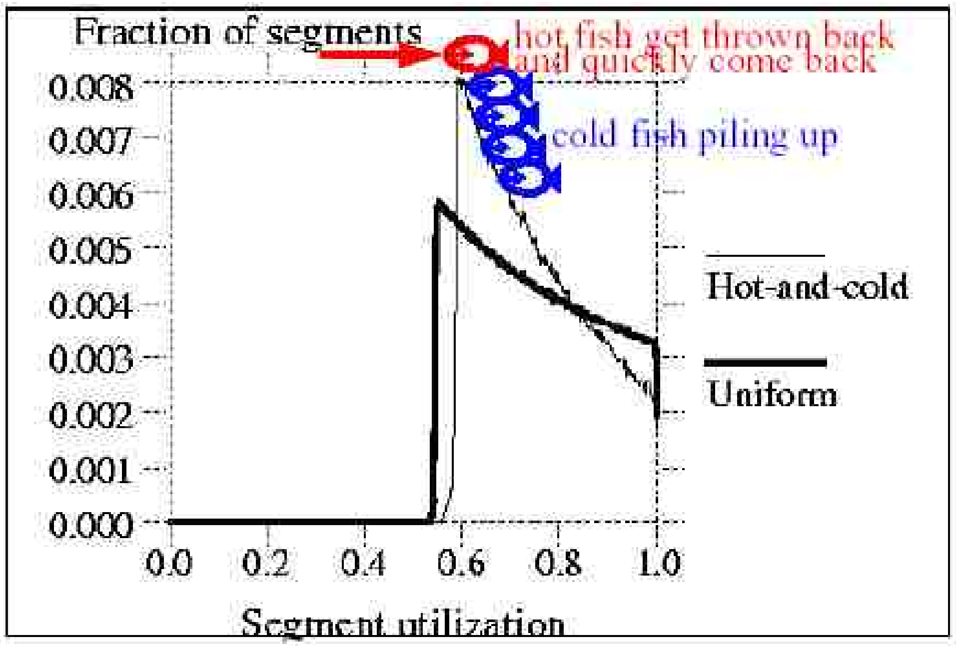
- Want bimodal distribution
  - clean low-utilization segments, easy
  - leave high-utilization segs untouched
- Workloads
  - random writes: still can do better than average u
  - typical file system has locality, can do even better

5.3 Greedy cleaner



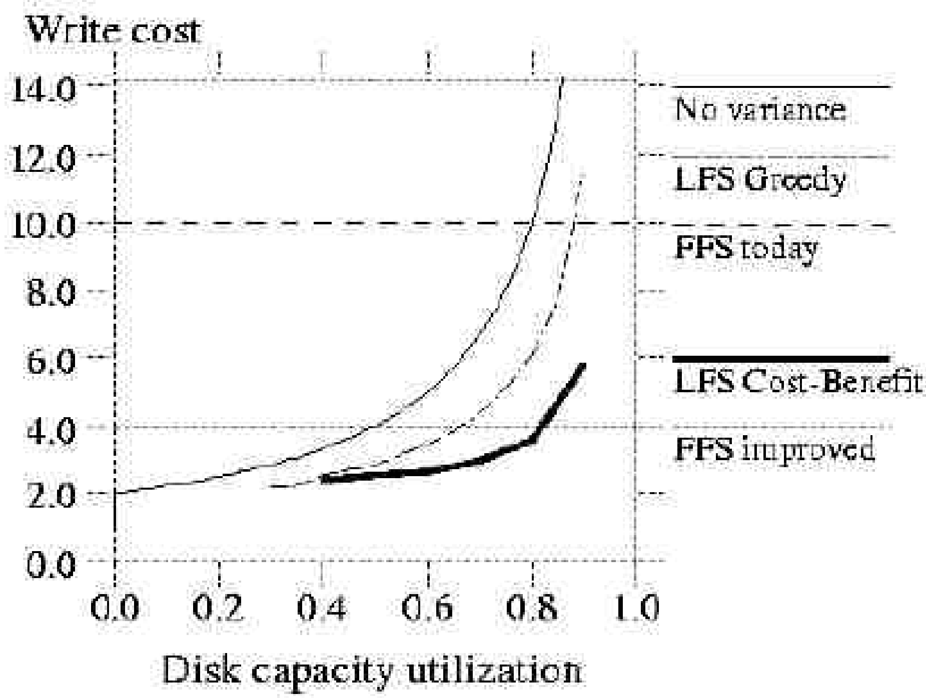
- Greedy cleaner: pick the lowest u to clean
- Works fine for random workload
- For "hot-cold" workload: 90% writes to 10% blocks
  - 1st mistake: not segregating hot from cold
  - Did that and it didn't help
  - Figure 4 shows the surprising result that locality and "better" grouping result in worse performance than a system with no locality!

5.4 What's wrong?



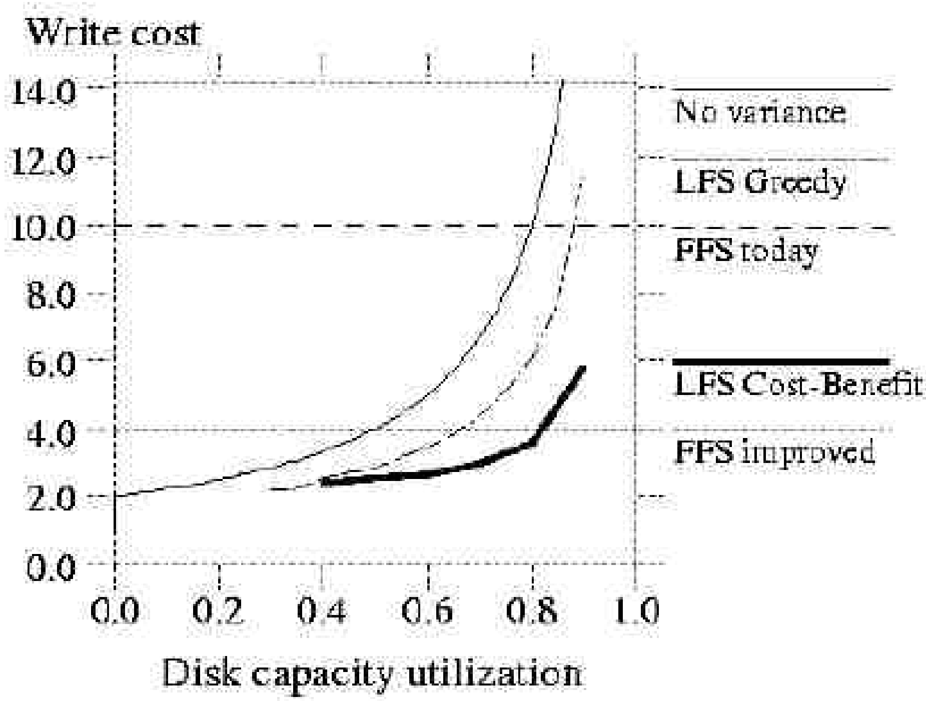
- Segments are like fish, swimming to the left
- Cleaner spends all its time repeatedly slinging a few hot fish back
- Cold fish hide lots of free space on the cliff but the cleaner can't get at them, and most fish are cold

5.5 Answer



- Cold free space more valuable: if you throw back cold fish, takes them longer to come back
- Hot free space is less valuable: might as well wait a bit longer

5.6 "Cost benefit cleaner"



- Optimize for benefit/cost = age*(1-u)/(1+u)
- Favors cold segments. Coldness of segment approximated by age of of the youngest block in the segment.
- Hot segments cleaned at 15% utilization, cold segments at 75%. Result is similar to generational garbage collection
- Segment usage table has the number of live bytes and the most recent modified time of any block.

5.7 Segment size?

An Example Followup Question

- What's the best segment size?
- Big: can amortize seek more effectively
- Small: even better chance to find segments that have low utilization, or even zero utilization
- Find the optimal compromise

5.8 Crash recovery

- Last sector written in checkpoint is the timestamp (giving atomic commit). Two checkpoint regions allow for a crash during a checkpoint.
- Checkpoint every 30 seconds
  - Checkpoint has addresses of all the blocks in the inode map and segment usage table, plus the current time and a pointer to the last segment written.
- Roll forward tracks changes to inodes, ignores new data blocks (why?)
- Directory operation log ensures consistency between inodes and directory entries. Directory operation entry is written before inode or directory entry.
- Directory operation log makes atomic rename easy. Why?
  - Directory entries cannot be modified while checkpoint is written.

6 Evaluation

6.1 Paper's conclusions

- Disk parameters
  - WREN IV disk – 1.3MB/s max BW, 17.5 avg seek, 300MB
  - LFS: 4KB block size, 1MB segment size
- Results
  - 10x performance for small writes
  - Similar large I/O performance
  - Terrible sequential read after random write. Temporal locality does not match logical locality.
  - Note:
    * 1990 disk Wren IV: 1.3MB/s BW, 17.5ms avg seek, 300MB storage
    * 2002 disk : 50MB/s BW (40x), 5ms avg seek (3x), 100GB storage (300x)
    * How change results?
    * How change design/parameters (segment size, checkpoint strategy, ...)
- Questions
  - Is LFS really as simple as FFS? Segment cleaning isn't.
  - Microbenchmark only? (Andrew benchmark 80% CPU)
  - How much is attributed to asynchrony? (Later work on delayed writes for metadata)
  - Story of impact of cleaning is simplistic?
  - I argued at start of discussion that this is example of good science. Still not perfect. What questions doesn't it answer?
  - How does read cost compare with FFS in practice? Is it OK to give up careful disk-physical-property-based placement and hope that read and write temporal locality will match?
- Other advantages
  - Fast recovery
  - Support of transactional semantics
  - Not necessarily an LFS monopoly though

6.2 Experimental evaluation

Note: I actually think they do a really good job overall. Still, let's see if we understand what they did and if there are any improvements...

6.2.1 Graph-by-graph analysis/critique

- Figure 3, 4, and 7 – Analytic model and simultion of write cost v. disk capacity utilization
  - Basic story –
    - 3: cleaning cost depends on utilization,
    - 4: cleaning cost depends on **minimum** segment utilization not avg
    - 4: But greedy cleaning does worse when there is locality (surprise!)
    - 7: Delay cleaning hot segments
  - Sanity check: Where does "FFS today = 10" come from? What about "FFS improved = 4"?

    X:
    - *Write = seek + rot + metadata write + seek + rot + data write + seek + rot + free space write*
    - *BW 1MB/s → write 1K in 1ms*
    - *Worst case – No locality: seek + rot = 15ms*
    - *Best case – locality: seek + rot = 0 (amortize across many writes to huge numbers of writes to nearby files...)*
    - *Range from 1/48 to 1/1 depending on workload*
    - *Paper uses microbenchmark experiment for "current case" – figure 8 "small file* **create***" – 10:1 to 50:1 advantage*
    - *(Should have used small file overwrite? Create may overstate benefits? Overwrite may be less advantageous b/c fewer seeks)*
    - *(Methodology question: Did Figure 8 use a single blocking thread? Could FFS get more throughput with multiple concurrent threads (allowing disk to schedule multiple outstanding requests?)*
    - *Paper extrapolates from Seltzer, Chen, and Ousterout "Disk scheduling revisted" to argue that best FFS could do is write cost of 4*
  - Figure 8: Small file performance
    - Basic story: LFS 10x faster for create/delete (and uses only 17% of disk BW)
    - What limitations, if any, from these experiments? How improve/expand on experiments? **X**:
      - *no cleaning → keep running long enough to get steady state performance with cleaning (vary free space)*
      - *create more expensive than overwrite → also run experiment with "overwrite" phase*
      - *Read performance is "best case" for LFS (same order as write) → try reads in random order*
  - Figure 9: Large file performance

    * Basic story: LFS modestly faster on sequential or random writes; LFS similar for sequential read after sequential write or random read after random write; *FFS faster for sequential read after random write*
  - Table II – production file system measurements of cleaning costs
    * Basic story: avg write cost 1.4 to 1.6 in production file system
    * (and this may be pessimistic – cleaning can be done in background?)

6.2.2 Higher level critique

Did they do the right experiments?

In what ways are experiments too generous to LFS? What is worst-case workload for LFS? **X**:

- *Microbenchmarks run with no cleaner*
- *Small file microbenchmarks have no concurrency; FFS* might *improve scheduling of writes with more concurrency*
- *Should have shown small writes not just small creates*
- *Worst case performance: random overwrite of small chunks of large file (e.g., transaction processing)*

In what ways are experiments too conservative about LFS's advantages? **X**:

- *"real world" cleaning costs may overstate cost; cleaner probably is able to do most of its work during periods of idleness; perhaps should have measured what fraction of cleaning done when idle...*

What questions are just not addressed? How could they be addressed? **X**:

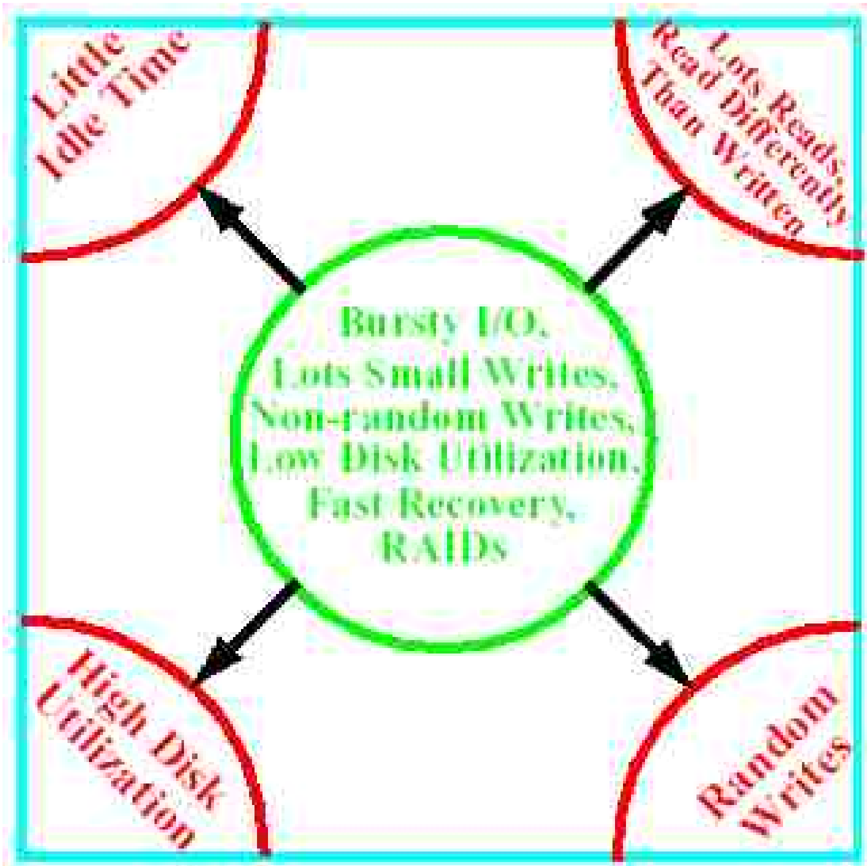
- *Real world performance. Does a user see improved real-world performance? (Ideas for testing – AFS benchmark, replay trace of real-world workload against LFS and against other production system, ...)*
- *Real world experience issue – cleaner grabs exclusive lock on everything – multi-second period during which everyone waits; implementation artifact, but still...*
- *Memory consumption – Seltzer93: "LFS [is] a very 'bad neighbor"' – LFS locks down 3 segments per file system plus buffers reserved for staging (64-128K per FS) and cleaner (avg 3.7MB/file system); if a system has 10 file systems mounted and 1MB segments → 60MB "locked down" by LFS (v. 32MB for the workstation they ran microbenchmarks on). Answer – (1) some of these are artifcats of implementation fixed by Seltzer93 (2) tech trends will reduce this (?)*

6.3 Ousterhout's (and Mike's) summary of meta-lessons

- Vary operating conditions and show each system both at its best and worst.
  - Mike: if you don't show the worst-case behavior of your system, someone else will
- Measure one level deeper than you publish; use your intuition to ask questions, not to answer them.
  - Mike: Think about graphs. Don't just say "up and to the right, that's what we expected." Be able to explain with back of the envelope calculations the magnitude of values; the slope of line.
  - Mike: Big danger in experimental systems: (1) guess answer, (2) run experiment (3) unexpected result (my system is not as good as the other system) $\rightarrow$ debug/tweak goto 2, (4) expected result (my system better than other) $\rightarrow$ done – as expected my system wins!
- Consider significance of results: all graphs should have a y-axis based at zero.
- Mike: Many complex heuristics in CS (FFS, TCP, ...) – how do we understand them? Danger – build "simple systems" that seem to work and then spend a decade or more figuring how why they work (perhaps a more systematic approach could be taken from the beginning...)

7 Conclusions

7.1 Why LFS? Why Not?



- LFS does well on "common" workloads
- LFS degrades for "corner" cases
- LFS architecture inherently flexible → easy to incorporate other FS paradigms

7.2 How radical is it?

- continuum
  - FFS: inodes: fixed, update-in-place; data: fixed, update-in-place
  - JFS: inodes: fixed, redo log; data: fixed, update-in-place
  - "Transactional FS": inodes: fixed, redo log; data: fixed, redo log
  - LFS: inodes: mobile, log+cleaner data: mobile, log+
- Compare "Transactional FS" v. "LFS"
  - In LFS need to be able to find data in log, but really no different than normal inode structure

- Compare cleaning cost v. replay cost
  - LFS: get to wait longer before cleaning $\rightarrow$ data may die
  - LFS: write cleaned data to log $\rightarrow$ fewer seeks
  - Transactional FS: wait shorter before re-write $\rightarrow$ don't have to read log (in common case)
  - TFS: Still get to batch many writes $\rightarrow$ maybe seeks are not too bad...

- Henson: LFS Failed because of segment cleaning overheads.
- Technology trends, solid state devices (SSD)
  - Reads are cheap, writes are expensive (large blocks).
  - LFS, but in device firmware!
  - Looks like a block device to OS, format with ext4.
  - Ooops, include TRIM command, which lets the file system tell the block device which blocks are free.