LFS

CS380L: Mike Dahlin

April 6, 2010

> Treat disks like tape.
>
> *J. Ousterhout*

1 Preliminaries

1.1 Review

1.2 Outline

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

1.3 Preview

2 Introduction

2.1 Why I teach this paper

- Not widely deployed
- Useful pedagogical tool – if you understand this, you understand any file system
- Good illustration of research methodology (both original work and subsequent evaluation)
- Useful ideas – ideas live on in – solid state drive file systems, solaris ZFS

2.2 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")

2.3 Technology trends

- Big memory(??)
  - ?? density curves of the disk trend paper
- High disk latency, ok bandwidth
- Disks becoming more complicated
- RAIDs and network RAIDs

2.4 Implications

- Reads taken care of (?)
- Writes not, because paranoid of failure
- Most disk traffic is writes
- Cant 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

2.5 Problems with UNIX FFS

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

2.6 Approaches

- Replace synchronous writes with asynchronous ones
- 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

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

- The log is the only and entire truth, theres nothing else

2.8 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?

3 Admin

midterm 1 grade distribution (fall 2004)

```
13 X
14 XXXX
15 XXX
16 XXX
17 XXX
18 X
19 X
```

Project checkpoint (due friday)

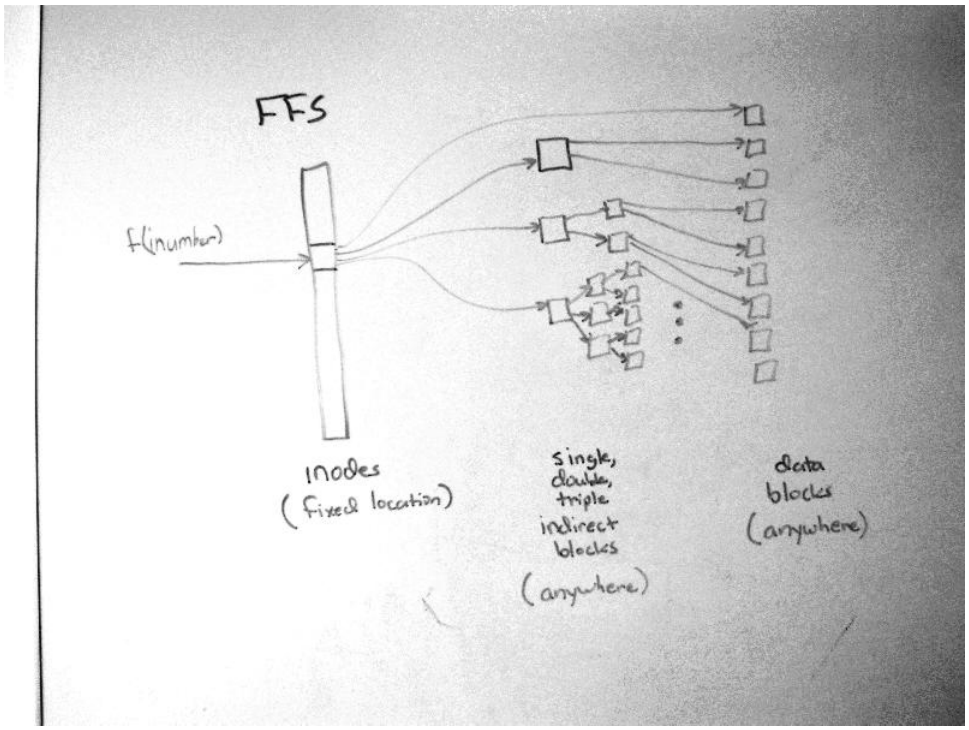
- Outline of final report
- High-level pseudo-code of design
- original milestones, progress, new milestones

Project design review (due friday)

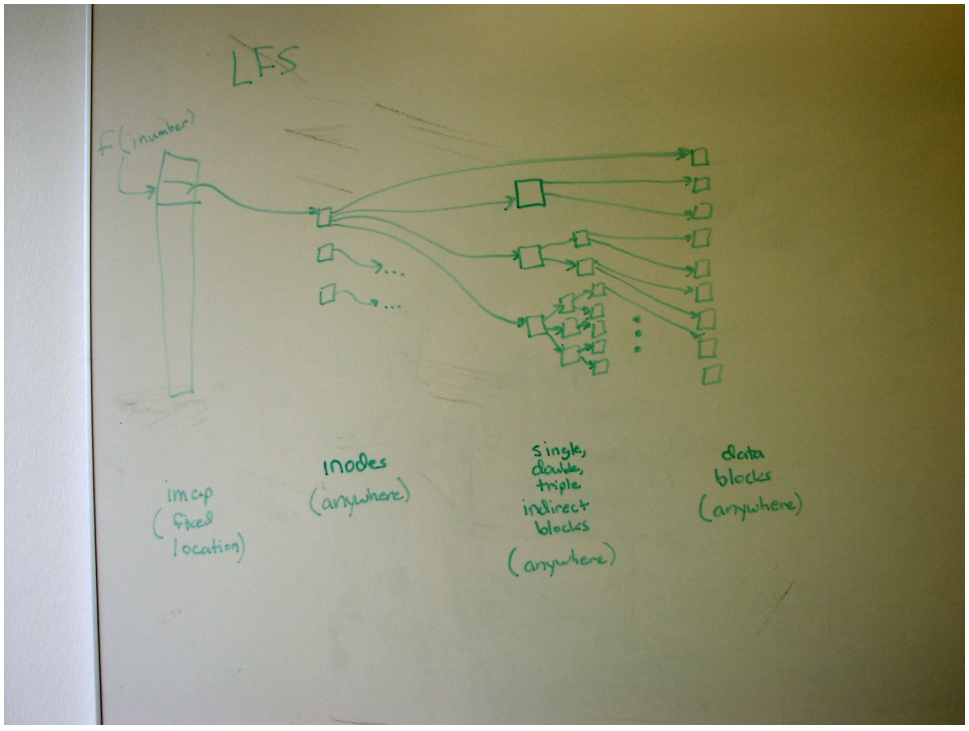
- 2 hr meeting with another group
- give them checkpoint (24hr in advance)
- present paper outline and pseudo-code
- Reviewers prepare report (see handout)

4 Index structures

4.1 Index structures in FFS



4.2 Index structures in LFS

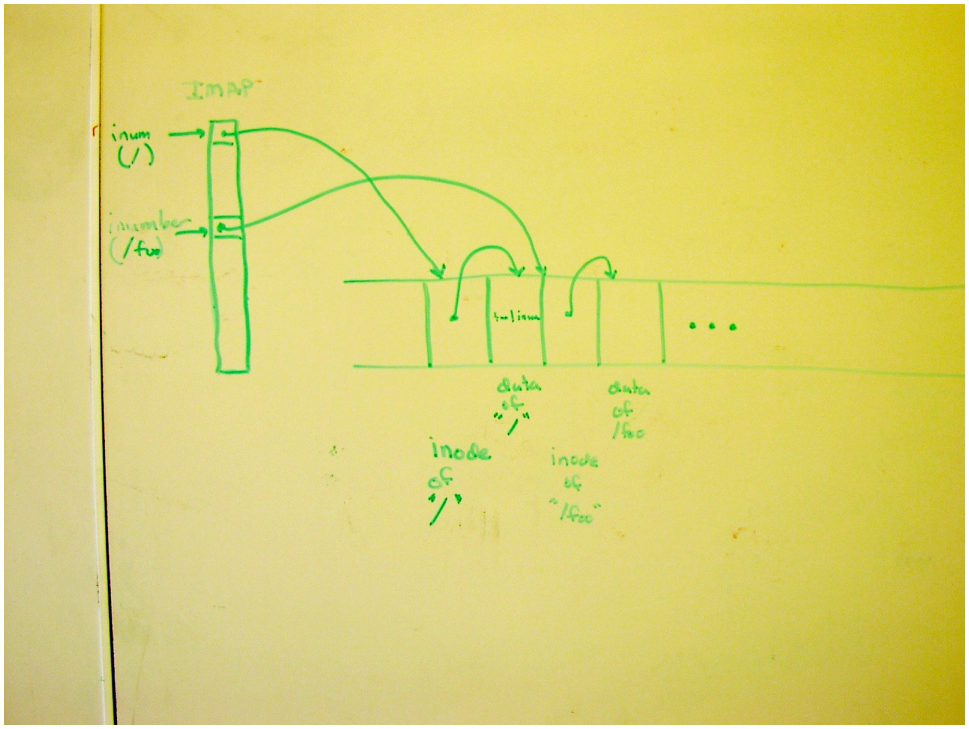


- Step 1 – move inodes to log
- Step 2 – find mobile inodes with fixed imap

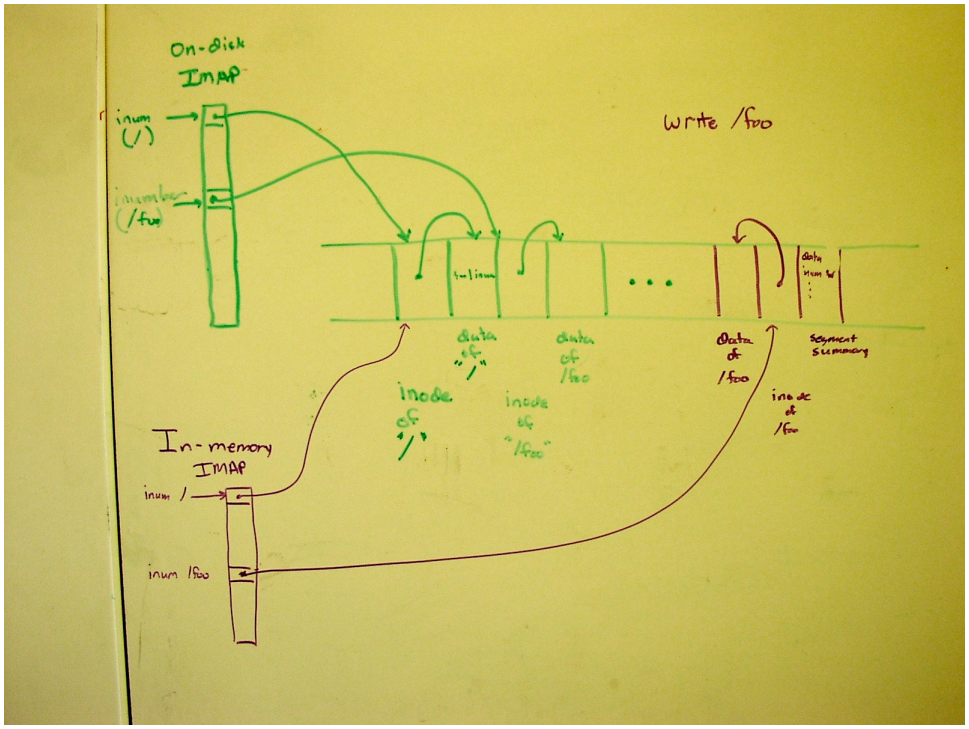
- Not obvious this is better
  - Why is this better?
  - Don't have to write imap after every write – just at checkpoints; otherwise roll forward
  - Couldn't you do this with original inode array?
  - Would there be any advantages to making imap mobile by adding another level of indirection?
- Compare 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%

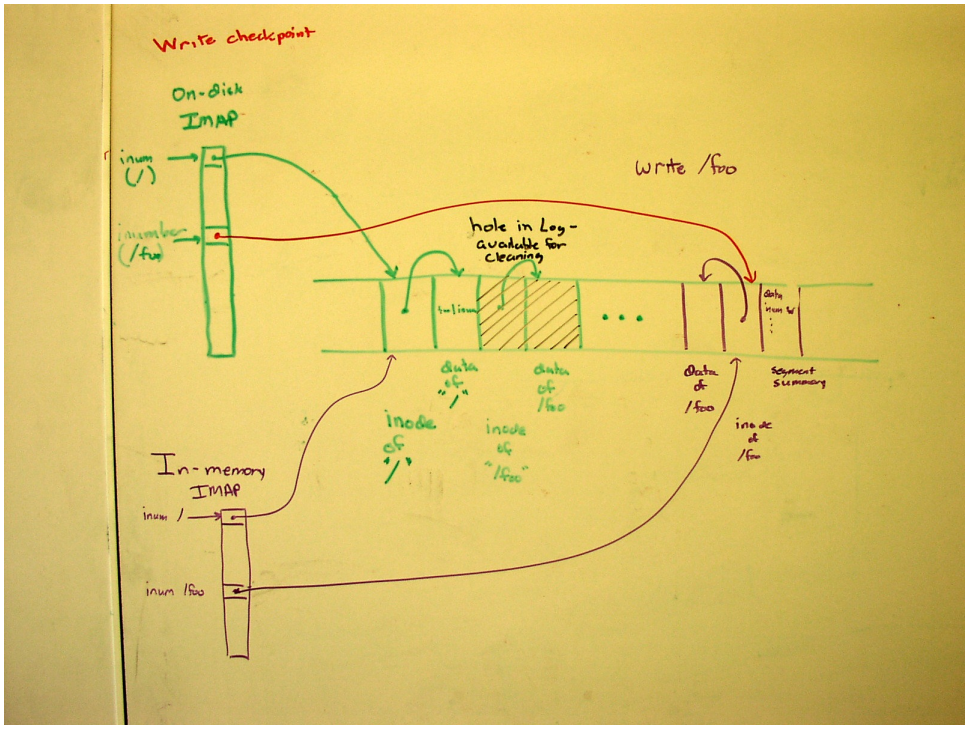
4.3 Example of LFS update



- Read "/foo"



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



- Update checkpoint (eventually)

5 Cleaning

How to get back free disk space

5.1 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

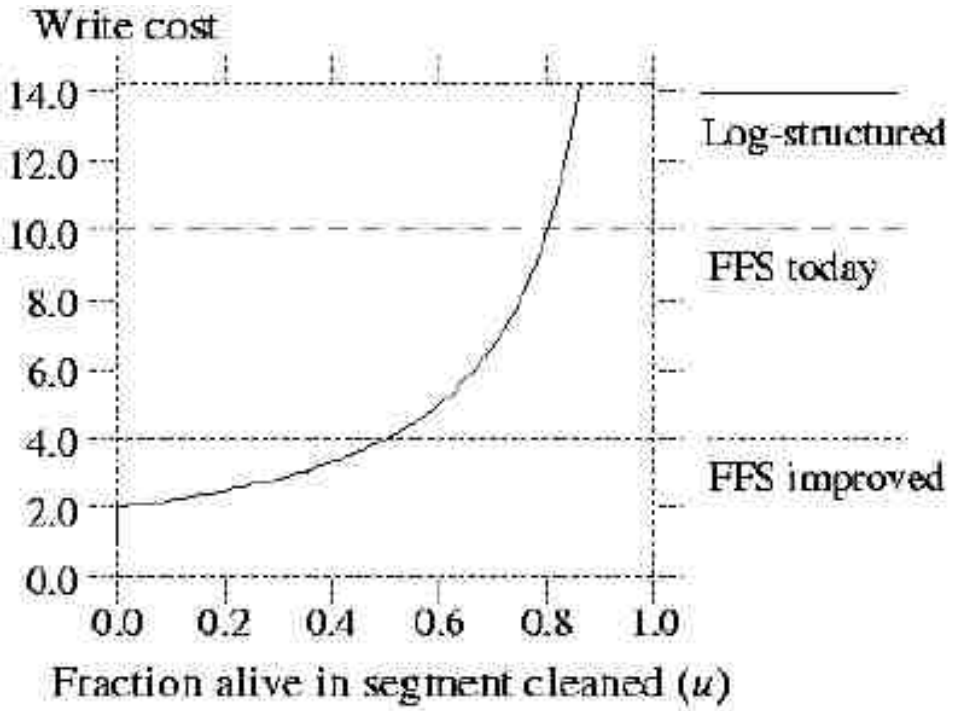
5.2 Option 2: Compact the log

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

5.3 Option 3: Segments: Combine threading + compaction

- Want benefits of both:
  - Compaction: big free space
  - Threading: leave long living things in place so I dont copy them again and again
- Solution: "semented 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)
- Many similarities with generational garbage collection?

6 Policies, evaluation

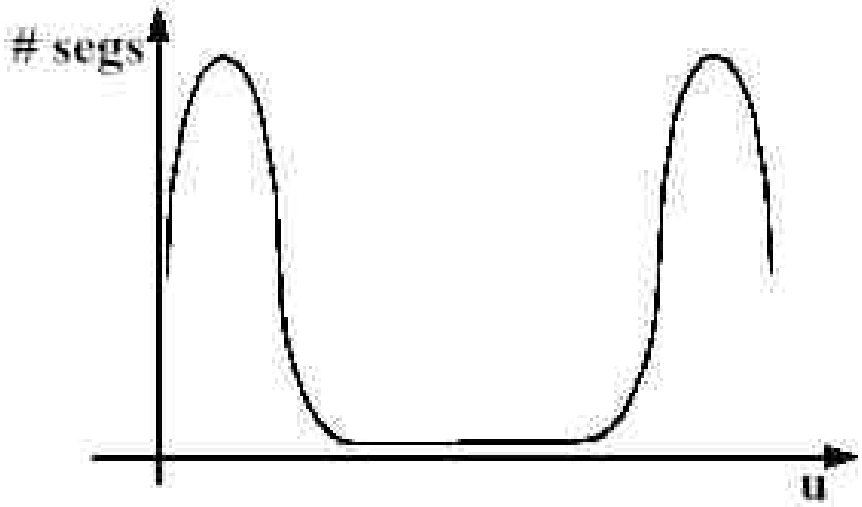


6.1 Is cleaning going to hurt?

- 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
- Conclusion: u better be small or its going to hurt bad
- A ha: u doesnt have to be overall utilization

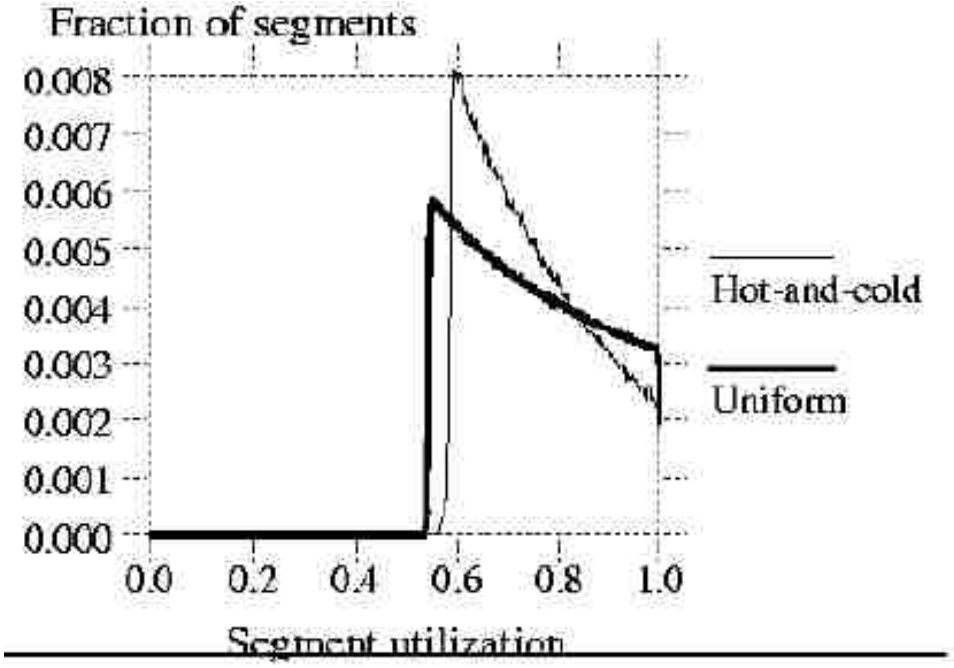
6.2 How to lower cost under utilization?

- Want



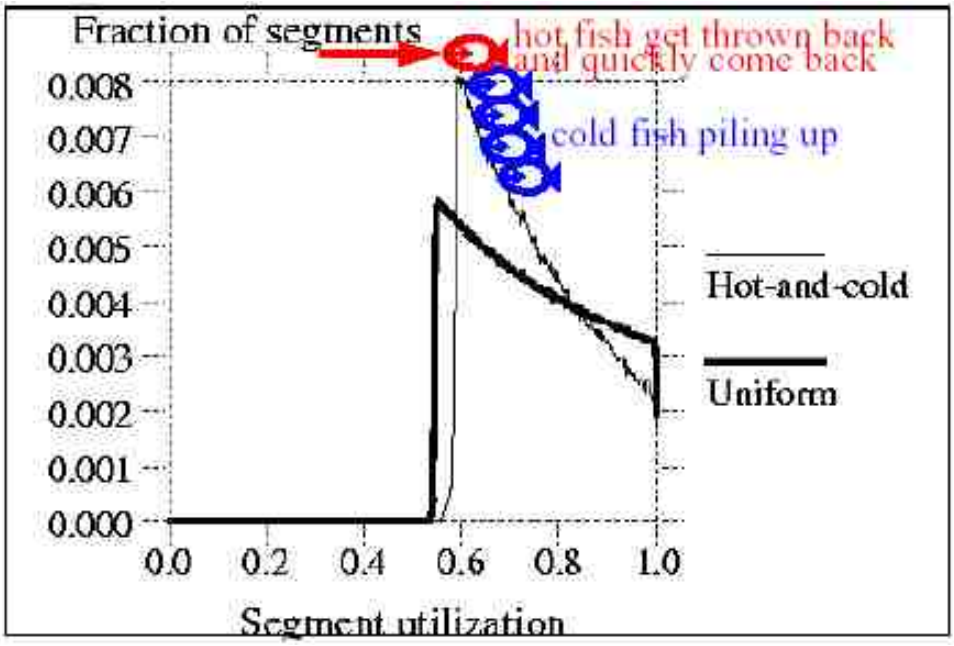
- bimodal distribution
  - clean low-utilization segments, easy
  - leave high-utilization segs untouched
- Workloads
  - random writes: still can do better than average u
  - typical file system has locality, can do even better

6.3 Greedy cleaner



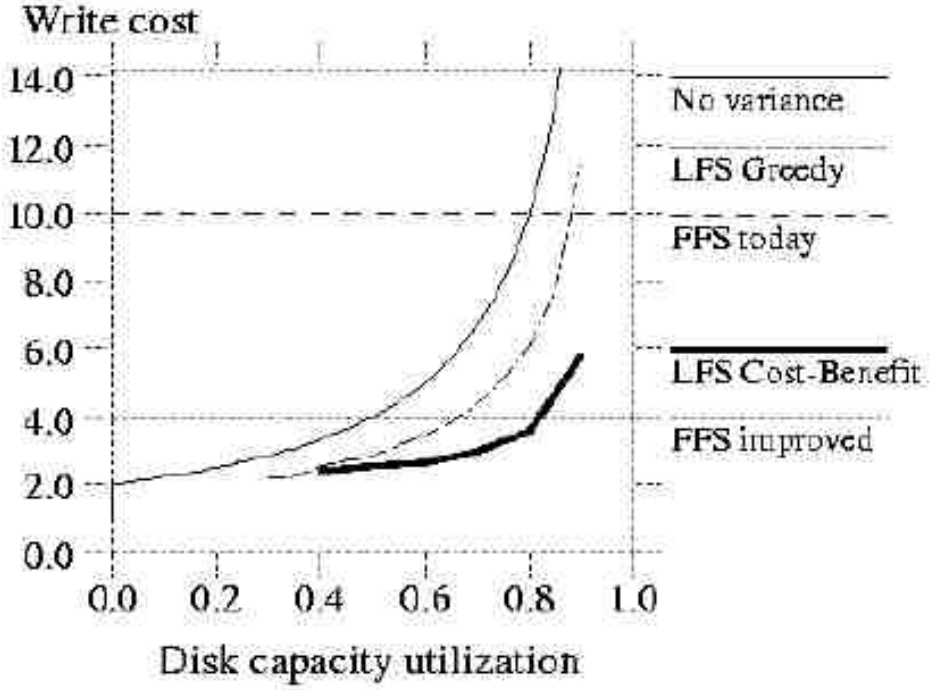
- Greedy cleaner: pick the lowest u to clean
- Works fine for random workload
- For "hot-cold" workload: 90% writes to 10% blocks
  - 1st mistake: not segregating hot from cold
  - Did that and it didnt help

6.4 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 cant get at them, and most fish are cold

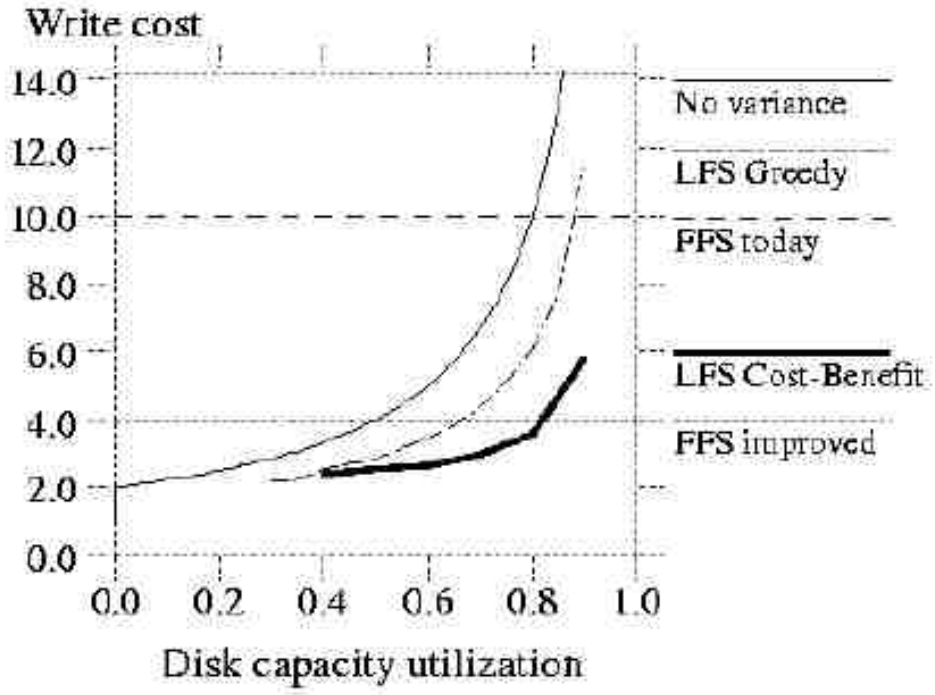
6.5 Answer



- 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

6.6 ‘

‘Cost benefit cleaner”



- Optimize for benefit/cost = age*(1-u)/(1+u)
- Favors cold segments

6.7 Segment size?

An Example Followup Question

- Whats 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

7 conclusion

7.1 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
  - 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
  - Microbenchmark only?
  - 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

7.2 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...

7.2.1 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"?
  - 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?
  - Figure 9: Large file performance
    * Basic story: LFS modestly faster on sequantial 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?)

7.2.2 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?
In what ways are experiments too conservative about LFS's advantages?
What questions are just not addressed? How could they be addressed?

7.3 Then the discussion begins...

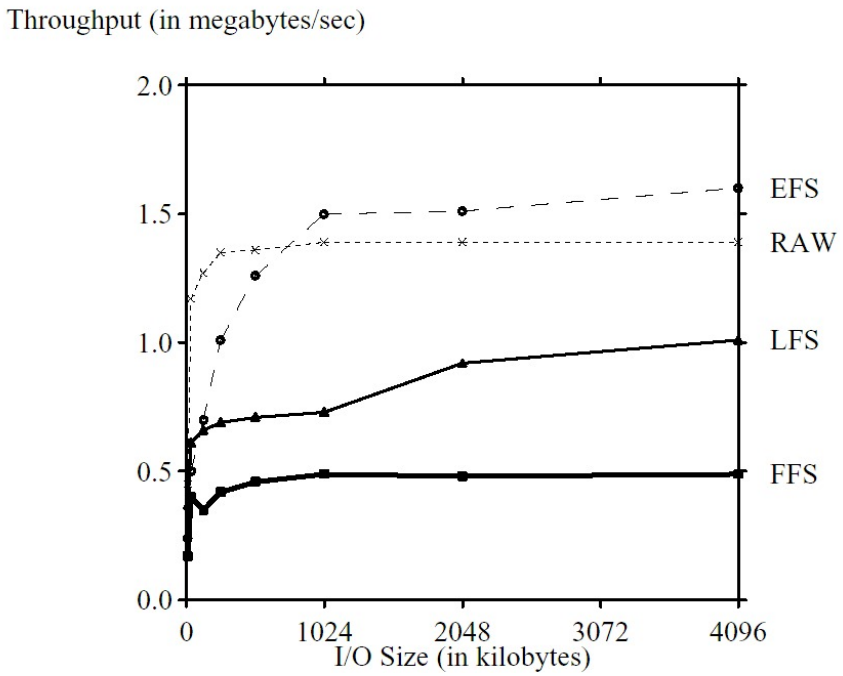
- Seltzer: USENIX BSD LFS papers reported unfavorable performance of LFS
- "An implementation of a log-structured file system for UNIX" Seltzer, Bostic, McKusick, Staelin USENIX 1993
- "File System Logging Versus Clustering: A Performance Comparison" Margo Seltzer, Keith A. Smith Harvard University Hari Balakrishnan, Jacqueline Chang, Sara ... USENIX 1995 http://www.usenix.org/publications/library/proceedings/neworl/full_papers/seltzer.pdf
- The web debate between Seltzer and Ousterhout
  - More info: http://www.eecs.harvard.edu/ margo/usenix.195/
  - Ousterhout's critique: http://home.pacbell.net/ouster/seltzer.html
  - Seltzer's response: http://www.eecs.harvard.edu/ margo/usenix.195/ouster.html
  - Ousterhout's second critique: http://home.pacbell.net/ouster/seltzer2.html
  - Ousterhout's comments on an earlier paper: http://home.pacbell.net/ouster/seltzer93.html
- Both sides made some valid points... but complicated because of philosophical debate on
  - Whats a "representative" workload? and
  - Where do you draw the line between fundamental flaw of the architecture and implementation artifacts?

7.4 Seltzer93

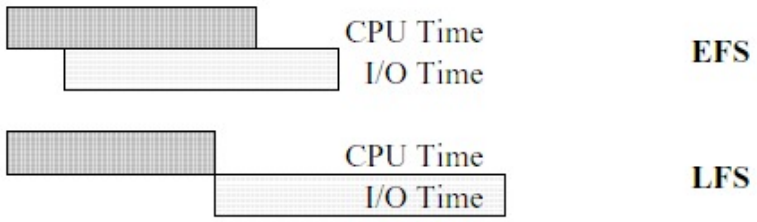
- Re-implement LFS on BSD
- Re-engineer to fix some issues (memory consumption, user-level cleaner, fsck, etc.)
- Compare LFS v. FFS v. EFS (Extent-based FS – FFS + some locality policies to improve large-file layout)
- Disk performance – HP 97560
  - avg seek 13.0ms
  - single rotation 15.0ms
  - track size 36KB
  - track buffer 128KB
  - disk BW 2.2MB/s
  - bus BW 1.6 MB/s
  - controller overhead 1.0ms
  - Track skew 8 sectors

- cylindar size 19 tracks
- disk size 1962 cylinders

- Figure 9 – maximum file system write bandwidth

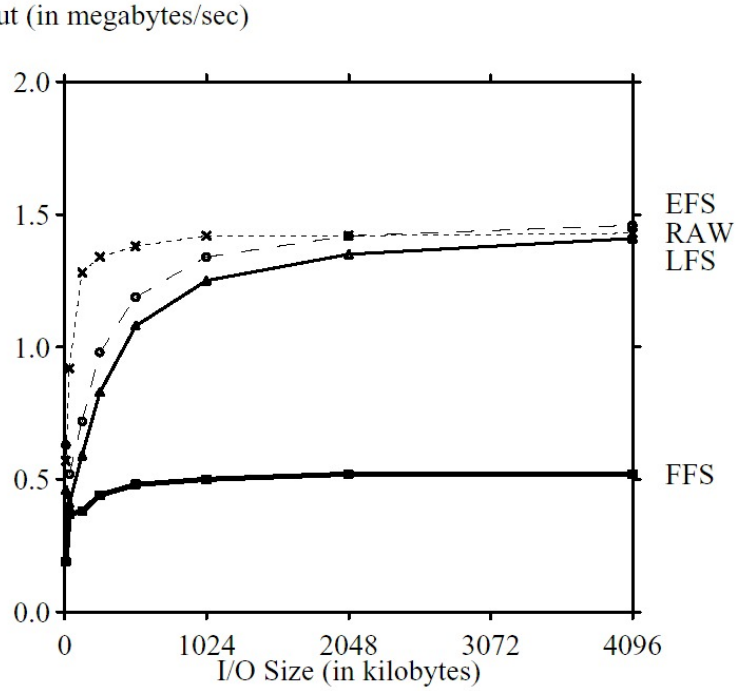


  - Benchmark: Create a file of size S and read or write entire file 50 times
  - Surprise 1: EFS beats raw disk BW
    * Explanation: raw writes are synchronous – setting up write i doesn't begin until write i-1 completes → disk is idle 5ms per whole-file write; EFS does async writes and keeps disk (actually the IO bus) busy
  - Surprise 2: LFS much worse than EFS and Raw for large writes
    * Text explains that this is an anomoly of benchmark – LFS delays *start* of write until 800KB have accumulated; LFS gets later start and therefore finishes later
    * Figure 10:



  - Conclude: EFS v. LFS have comparable write bandwidth; nearly 100% of raw BW. "When individual write response time is an issue, LFS incurs a performance penalty due to its delayed write policy"

- Figure 11 – maximum file system read bandwidth



  - Read a file of size S 50 times
  - Conclude: LFS and EFS have similar read bandwidth

rformance for medium files

- Table 6 – Modified andrew benchmark results

	Phase 1 Create Directories	Phase 2 Copy Files	Phase 3 Stat Touch Inodes	Phase 4 Grep Touch Bytes	Phase 5 Compile	Total
FFS	2.10 (0.30)	7.90 (0.30)	6.30 (0.46)	9.00 (0.00)	44.80 (0.40)	70.1 (0.70)
EFS	2.10 (0.30)	7.90 (0.30)	6.70 (1.19)	9.10 (0.30)	44.40 (0.49)	70.2 (1.60)
LFS	0.33 (0.47)	5.00 (0.00)	6.50 (0.81)	9.07 (0.25)	42.90 (1.40)	63.8 (2.34)
LFSC	0.43 (0.49)	5.09 (0.28)	6.37 (0.48)	9.07 (0.26)	42.61 (0.49)	63.6 (0.62)

Table 6: Single-User Andrew Benchmark Results. This table shows the elapsed time for each phase of the benchmark on each file system. Reported times are the average across ten iterations with the standard deviation in parentheses. LFSC indicates that the benchmark was run on the log-structured file system with the cleaner running, but the similarity in results for most phases indicates that the cleaner had virtually no impact on performance. Overall, LFS demonstrates approximately a 9% difference in performance which can be attributed to asynchronous file creation and write-clustering.

  - Benchmark: create directories, copy files, stat files, touch all bytes, compile files
  - Conclude: "Overall LFS demonstrates a 9% improvement over EFS and FFS...the difference is isolated to phases one, two, and five."

- Table 7 – TPC-B performance results

	Transactions per second	Elapsed Time 1000 transactions
FFS	14.2	70.23 (1.0%)
EFS	16.8	59.51 (2.1%)
LFS (no cleaner)	19.3	51.75 (0.6%)
LFS (cleaner, 1M)	11.6	85.86 (5.3%)
LFS (cleaner, 256 K)	12.4	80.72 (1.8%)

  - Benchmark: Read and write random small records in a large file; 80
  - Conclude: When cleaner not running LFS beats EFS by 15%
  - Conclude: When cleaner running EFS beats LFS by 22%

- After reading this study, should one conclude that LFS's microbenchmark performance is comparable to EFS's (and sometimes 2x worse???) and that LFS's application-performance ranges from 10% better (andrew) to 20% worse (TPC)?

- Ousterhout's critique ("A Critique of Seltzer's 1993 USENIX Paper" J. Ousterhout)

  - "The comparisons between BSD-LFS, FFS, and EFS suffer from three general problems: a **bad implementation** of BSD-LFS, a **poor choice of benchmarks**, and a **poor analysis** of the benchmarks. Combined together, these problems invalidate many of the paper's conclusions; LFS is a much better file system architecture than this paper suggests." (Emphasis added)
  - How should we evaluate the choice of benchmarks? What properties should a good set of benchmarks have?
  - Are there any ways in which the benchmarks chosen make LFS look less attractive than it might be? (Conversely, are there ways in which Rosenblum and Ousterhout's benchmarks in the 1991 paper make LFS look more attractive than it might be?)
  - Ousterhout claims the following bugs affect the results. Are they bugs (or justifiable implementation trade-offs)? How would they affect the results?
    * The system did not implement fragments, so the smallest block size was 4KB or 8KB, compared to 512 or 1024 bytes for EFS and FFS. This resulted in an unnecessary 4-8x reduction in LFS performance for small files.
      · How might this affect results?
    * BSD-LFS contained a bug causing it to layout segments backwards on disk. This damages read performance during sequential reads by wasting almost a full disk rotation between each block of a file and the next block. The performance penalty is most noticeable for medium-sized files.
      · How might this affect results?
    * BSD-LFS stores the access time in inodes, rather than putting it in the inode map as in LFS. This means that the inode must be rewritten each time the file is read, so the inode tends to migrate away from the file on disk, causing long seeks during read accesses.
      · How might this affect results?

* BSD-LFS flushes indirect blocks and inodes to disk unnecessarily, resulting in excess writes and increased cleaner overhead (this problem is described in more detail in my critique of the 1995 USENIX paper).
  - How might this affect results?
* The explanation for Figures 9 and 12 doesn't discuss the impact of a 56-Kbyte I/O limit, which accounts for most of the performance differences at large transfer sizes.
  - Seltzer's on-line response explains this issue in more detail "The maximum transfer unit between memory and the disk is 56 KB. The FFS places a rotational delay between 56 KB units; LFS does not. Therefore, when data is being written to disk, LFS loses a rotation between 56 KB writes while FFS loses only a rotdelay (approximately 1/4 revolution)."
  - How might this affect results?

7.5 Seltzer95

- "File System Logging Versus Clustering: A Performance Comparison" Margo Seltzer, Keith A. Smith Harvard University Hari Balakrishnan, Jacqueline Chang, Sara ... USENIX 1995 http://www.usenix.org/publications/library/proceedings/neworl/full_papers/seltzer.pdf
- Figure 1/2: Validation of BSD-LFS small/large file performance

Fig 1 (small):

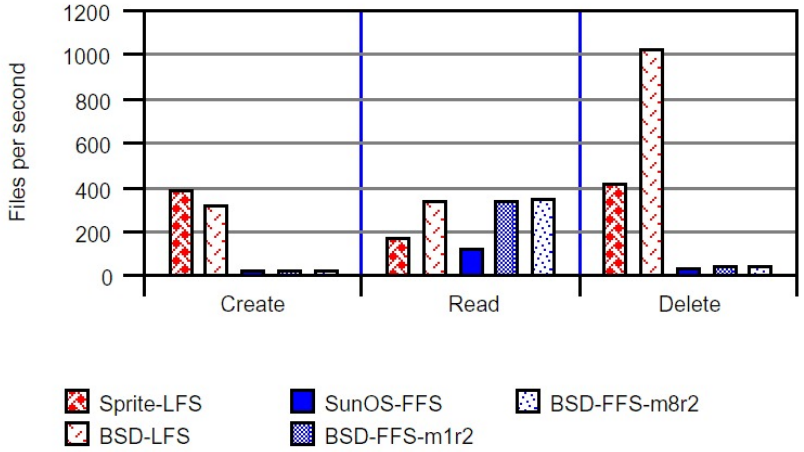
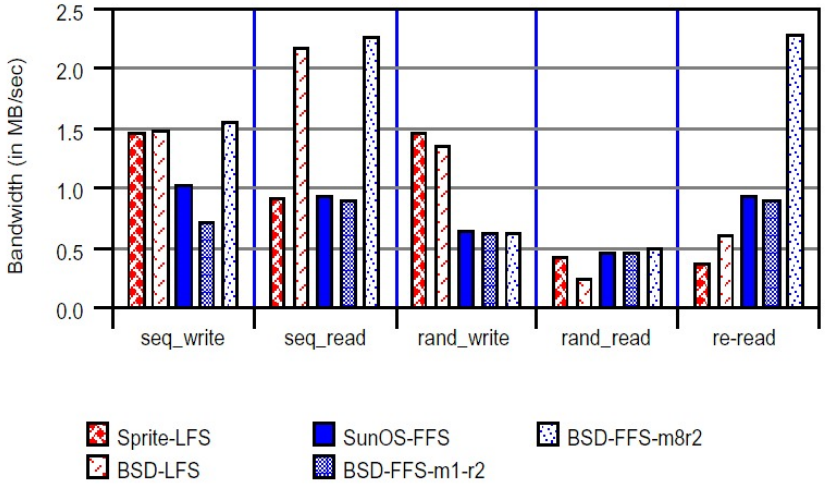


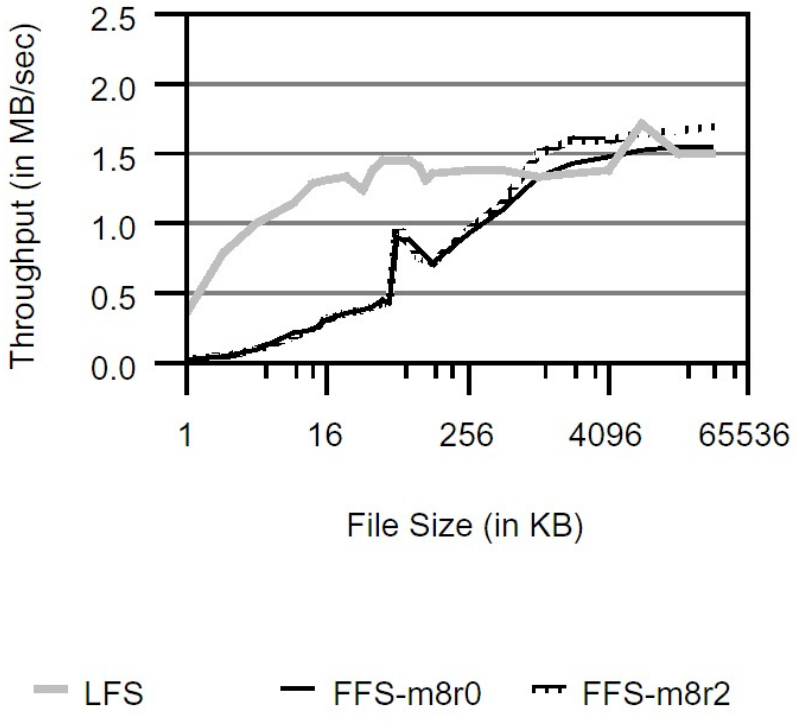
Fig 2 (large):



  - Goal: Verify that BSD implementation is comparable to Sprite implementation (e.g., make sure no big performance bugs)

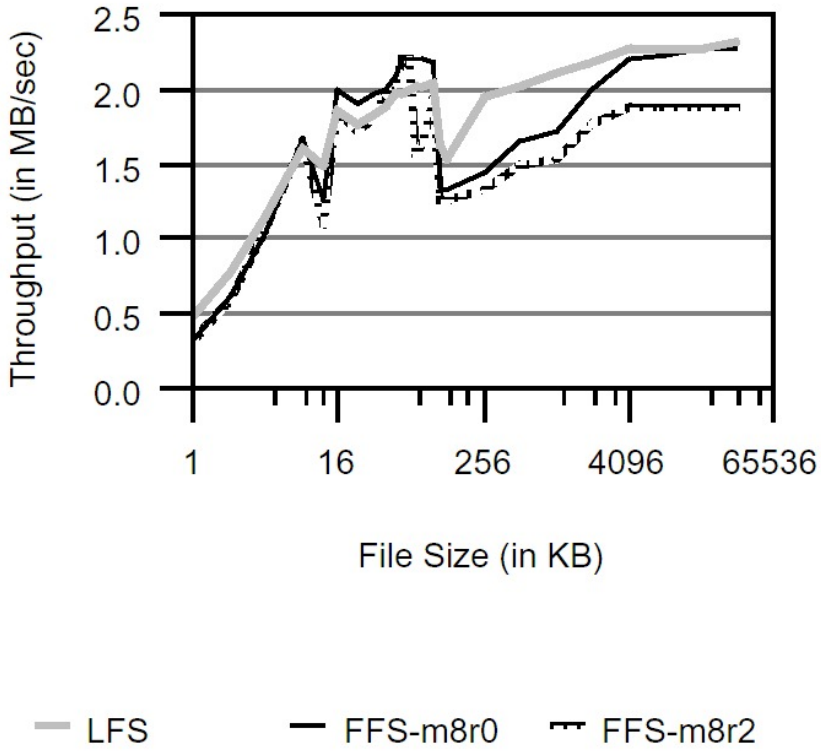
  - Repeat Rosenblum's experiments
  - Scale results by disk BW, disk seek, and CPU performance
  - Expect – BSD and Sprite LFS match; BSD and Sprite FFS match
  - Results: Same basic conclusions; differences from different track-buffer sizes, read-ahead policies, and skip-sector positioning

- Figure 2: Validation of BSD-LFS large file performance
- Figure 3: Create performance v. file size



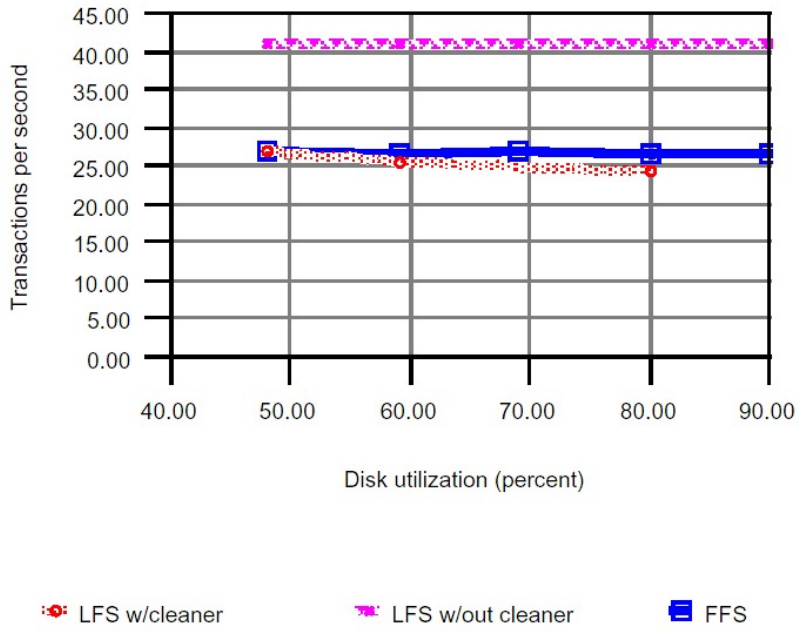
  - Multiple files – at least 32 MB of data or at least 10 files; for small files 100 files per directory
  - Conclude: LFS has 2 advantages: (1) async writes ("In contrast, each time a creat system call returns, FFS guarantees that the file has been created and will exist after a system crash.", (2) sequential layout
  - (notice log-scale x axis)

- Figure 4: Read performance v. file size



  - FFS's problem for medium-sized files b/c (1) rotdelay gap for "sequential" layout fragments subsequent files "the ability to achieve the superior write performance of the FFS-m8r2 in the create test is precisely the limiting factor in the read case", (2) "FFS begins allocation in a new cylindar group when the first indirect block is added"

- Figure 5 and 6: Overwrite/Delete perf v. file size
  - Overwrite: LFS has significant advantage for files up to 128KB
  - Delete: LFS has significant advantage for all file sizes
- Figure 7: Transaction processing performance



- Main table: 1,000,000 100 byte records (237 MB file with indexing)
- 3 other tables totalling less than 200KB
- Each transaction: random account record is read and updated; 3 more updates (one to each other table)
- Seltzer et al's conclusions:
  - In the absence of the cleaner, LFS provides approximately 50% better performance than FFS. The 50% performance difference can be attributed to LFSs ability to perform the random writes as sequential writes. In the LFS case, as dirty pages are evicted from the user-level buffer cache, they are copied into the file system cache. The dirty blocks remain in the cache until the number of dirty blocks exceeds a write threshold and LFS triggers a disk write. With the current system configuration, this triggering occurs when 115 blocks have accumulated (representing 115 transactions). These transactions progress at a rate limited by the time required to randomly read the account records from disk. To read, we must copy the page being read from the kernel into the user cache and must also evict a page from the user cache, copying it into the kernel. On our system, these two copies take approximately 1.8 ms. With a 9.5 ms average seek, a 5.5 ms average rotational delay, and a 1.6 ms transfer time, each random read requires 18.4 ms for a throughput of 54 transactions per second. Next, the segment must be flushed. The 115 data blocks are likely to have caused the 58 indirect blocks in the account file to be dirtied, so our segment contains 115 data blocks, 58 indirect blocks, one inode block, and one segment summary for a total of approximately 700 KB. Using the bandwidth numbers from Section 3.3, we can write the 700 KB at a rate of 1.3 MB/sec for a total time of 0.5 seconds. *Therefore, processing 115 transactions requires 115 x 18.4 + 500 ms yielding 44 transactions per second, within 7% of our measurement.* (Emphasis added.)
  - The calculation for FFS is much simpler: throughput is limited by the performance of the random reads and writes. Each random I/O requires a 9.5 ms seek, a 5.5 ms rotation, a 0.9 ms copy, and a 1.6 ms transfer for a total of 17.5 ms yielding throughput of 28.6 transactions per second, within 7% of our measurement.
  - As discussed earlier, LFS fills segments at a rate of 115 transactions per 700 KB or 168 transactions per segment. For simplicity, call the database 256 MB and the disk system 512 MB. This

requires 256 segments, 43,000 transactions, or 1000 seconds at the no-cleaning LFS rate. After the 1000 seconds have elapsed, LFS must clean. If we wish to clean the entire disk, we must read all 512 segments and write 256 new ones. Let us assume, optimistically, that we can read segments at the full bus bandwidth (2.3 MB/sec) and write them at two-thirds of the disk bandwidth (1.7 MB/sec), missing a rotation between every 64 KB transfer. The cleaning process will take 223 seconds to read and 151 seconds to write for a total of 374 seconds. Therefore, at 50% utilization our best case throughput in the presence of the cleaner is 31.3 transactions per second. This is within 15% of our measured performance. Unfortunately, LFS cannot clean at the optimal rate described above. First, the transaction response would be unacceptably slow while the cleaner stopped for six minutes to clean. Secondly, the calculations above assumed that the disk is read sequentially. ... Since LFS cannot clean at its maximal rate, it should clean at a rate that permits it to perform its segment reads and writes at near-optimal speed. At 50and produce one clean segment. Reading one megabyte requires a random seek (9.5 ms) onehalf rotation (5.5 ms) and a 1 MB transfer (435 ms) for a total of 450 ms per segment read. Rewriting the segment requires the same seek and rotation, but the transfer requires 588 ms for a total of 603 ms for the write or 1.5 seconds to clean the segment. In the steady state, this cleaning must be done for each 168 transactions. Our throughput without the cleaner is 41 transactions per second, so it takes 4.1 seconds to execute 168 transactions and 1.5 seconds to clean, yielding 5.6 seconds or 30.0 TPS. This is within 10number.

- Ousterhout's critiques of 1995 paper
  - "The measurements in Section 4 are counter-intuitive, and the back-of-the-envelope calculations used to support those measurements are incorrect. At present neither Seltzer nor I can explain the measurements."
    - Ousterhout argues that Seltzer's calculations omit variance of utilization across segments, so cleaner will be able to pick less-heavily-utilized-than-average segments; He re-does the calculations and gets 34.4TPS – 27% higher than measured ("not 10% as reported in the paper").
  - "BSD LFS does not include a simple optimization that should substantially improve the LFS results in Section 4."
    - "The bug is that BSD-LFS writes all dirty indirect blocks whenever it writes any dirty data blocks. In this benchmark, 58 out of every 173 blocks written to disk are indirect blocks. The indirect blocks get modified almost immediately after being written, so they result in a lot of free space on disk that must be reclaimed by cleaning.
      "I believe that the best policy is to write dirty indirect blocks for a file only if the file has been closed or if the LRU replacement policy determines that the block should be replaced in the cache. In the normal case where a file is opened, written sequentially, and immediately closed, this guarantees that the indirect blocks will be near the file's data on disk. In the case where a file is open a long time and is being accessed randomly, as in the transaction processing benchmark, there is nothing to be gained by writing the indirect blocks. They are not needed for reliability, since there is already enough information in the log to recover the data blocks after a crash without the indirect blocks. If a file is open there will probably be more I/O to the file soon, so the indirect blocks are likely to be modified soon, which invalidates any copies on disk. Writing the indirect blocks just wastes write bandwidth and makes more work for the cleaner."

7.6 Seltzer's conclusions (1995 paper)

- "LFS is an order of magnitude faster on small file creates and deletes."

  - Seltzer et al attribute this to two factors: (1) Sequential layout and (2) asynchronous implementation; they note that (2) can also be gained by journaling

- "The systems are comparable on creates of large files (one-half megabyte or more).
- "The systems are comparable on reads of files less than 64 kilobytes.
- "LFS read performance is superior between 64 kilobytes and four megabytes, after which FFS is comparable.
- "LFS write performance is superior for files of 256 kilobytes or less.
- "FFS write performance is superior for files larger than 256 kilobytes.
- "Cleaning overhead can degrade LFS performance by more than 34% in a transaction processing environment.
- "Fragmentation can degrade FFS performance, over a two to three year period, by at most 15% in most environments but by as much as 30% in file systems such as a news partition.
  - Note that Ousterhout believes this FFS estimate is optimistic and that the actual cost is higher

7.7 Ousterhout's conclusions (on-line critique)

"If the original paper by Rosenblum and myself is combined with all of Seltzer's data, including both the data in the two USENIX papers and the data in her web response, I think it is reasonable to draw the following conclusions:

- "It is possible to find particular benchmarks and system configurations where either file system dominates the other.
- "Considering typical usage patterns but ignoring the costs of cleaning in LFS and fragmentation in FFS, LFS performance is almost never worse than FFS and often substantially better. In the best case for LFS (writes of small files) LFS is 4-10x faster; in the worst case (writes of large files) LFS is about 5% slower.
- "Fragmentation can degrade actual FFS read performance by 15% and write performance by 25%, while cleaning costs effectively add 20-60% to the cost of writes in LFS without affecting the cost of reads. In the worst case, fragmentation degrades overall FFS performance by about 35-45% while cleaning degrades overall LFS performance by about 45
- "LFS performance for transaction processing workloads ranges from about the same to about 10% worse than FFS, depending on the disk utilization.
- "The BSD-LFS implementation is still quite immature relative to FFS, so its performance is likely to improve relative to FFS as optimizations like the one described above [delayed indirect block writes] are implemented.

"It's also important to remember some of the other advantages of LFS that weren't addressed in Seltzer's measurements, such as faster crash recovery and the ability to handle striped disks and network servers much more efficiently than FFS. Overall, the available data suggests that LFS is a much better file system than FFS. "

7.8 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) → debug/tweak goto 2, (4) expected result (my system better than other) → done – as expected my system wins!
- Consider significance of results: all graphs should have a y-axis based at zero.
  - Mike: Personal pet peeve...
- 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...)

8 Conclusions

8.1 Why LFS? Why Not?

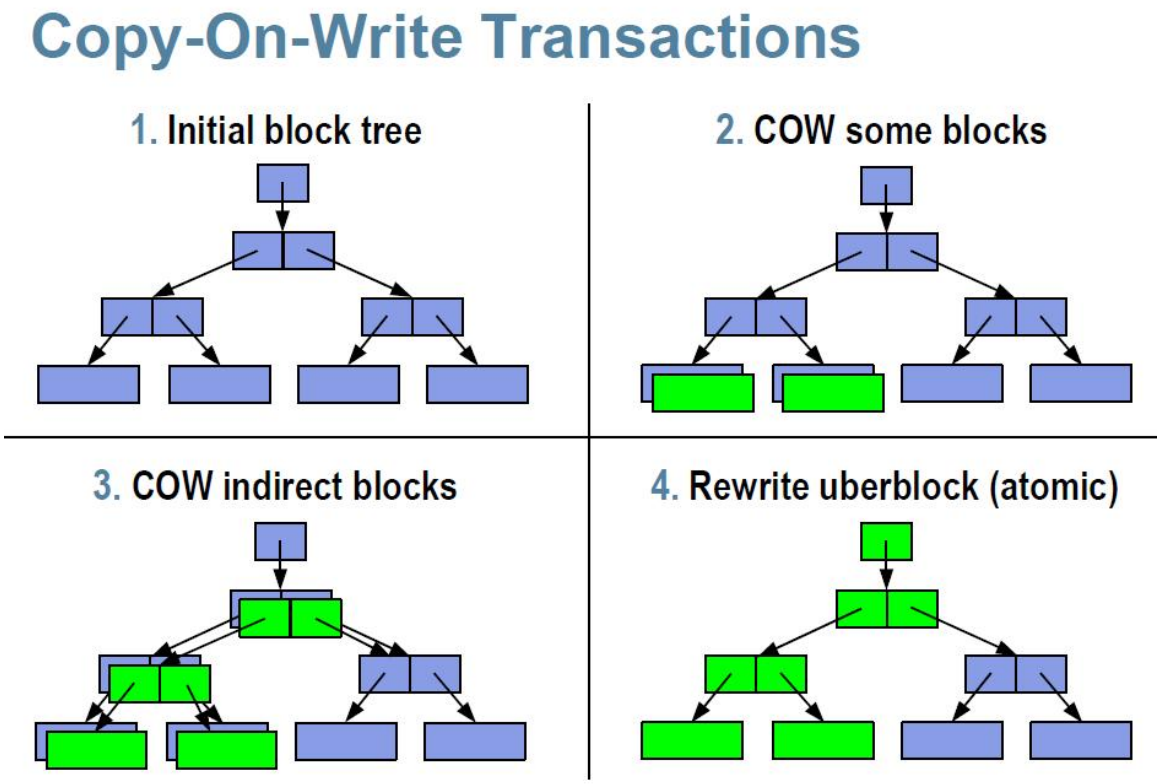
- LFS does well on "common" workloads

- LFS degrades for "corner" cases
- LFS architecture inherently flexible → easy to incorporate other FS paradigms

8.2 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 → data may die
    - LFS: write cleaned data to log → fewer seeks
    - Transactional FS: wait shorter before re-write → don't have to read log (in common case)
    - TFS: Still get to batch many writes → maybe seeks are not too bad...
  - Answer: idunno

9 ZFS

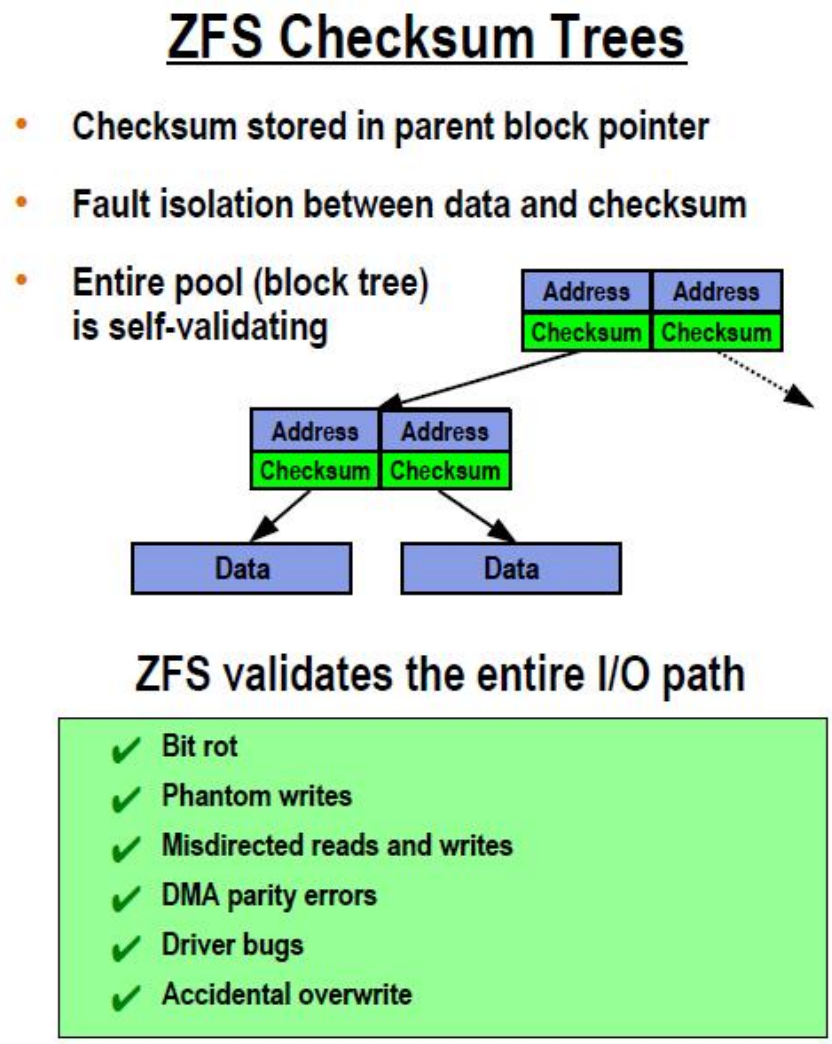


- Still have not found good reference on ZFS technical details
- Slides above hint that ZFS might be like LFS but with "threaded log" instead of segments

- How would you do LFS with threaded log?
  - Don't you give up a lot w/o segments (wasn't whole point to do long sequential writes?

    Mike: *Mechanism v. policy – mechanism allows writes with low locality, but your policy might still try to do long sequential runs when such spaces available; add an optional defragmenter instead of requiring a cleaner.*
  - How do you get transactional multi-sector update without sequential log?

    Mike: *Write of uberblock is "commit" and pointers tell you what is committed*
  - Nice additional optimization: checksums



  - How do you do crash recovery w/o log (to find recent writes) or segment summary to say what is inside of each block?

    Mike: *No need for recovery, atomic commit every update as new checkpoint*
  - How does compactor know what blocks are free and which are live (in LFS segment summary gives mapping from block number to inumber/offset; how do you do that w/o segment summary?

    Mike: *Can just have a bitmap that gets updated atomically. Store bitmap in a file ("file 13"). Could also store "segment summary" info in a special file for the entire disk (instead of per segment have block num to inum/offset mapping fror whole disk)*

- Any optimizations that LFS can do that threading give up?

  Mike: *ousterhout argues that you don't want to write inodes and indirect blocks with each update; lose that chance here, I think*

- Do the performance results in ZFS v. ext3 paper now make sense?

10 NAND file systems

- Challenge
  - Physical reality – fast read (no moving part), write is two step: (1) clear 512KB region (SLOW), (2) write 1KB block
  - Physical reality – limited lifetime (max number of "clear" operations) → need to "wear balance" disk by spreading writes
  - Commercial reality – need to export "disk" interface ("read/write(sector number)")
- Observation: Looks kinda like LFS...
- block IDs instead of file/offset – how to design index structure?
- How to do cleaning, wear leveling?
- ...