Log Structured FS

Arvind Krishnamurthy
Spring 2004

Log Structured File Systems

- Radical, different approach to designing file systems
- Technology motivations: some technologies are advancing more faster than others
  - CPU are getting faster every year (x2 every 1-2 years)
  - Everything else except CPU will become a bottleneck (Amdahl's law)
  - Disks are not getting much faster
  - Memory is growing in size dramatically (x2 every 1.5 years)
  - File systems ➔ File caches are a good idea (cut down on disk bandwidth)

Motivation (contd.)

- File System motivations:
  - File caches help reads a lot
  - File caches do not help writes very much
  - Delayed writes help but cannot delay for ever
  - File caches make disk writes more frequent than disk reads
  - Files are mostly small -- too much synchronous I/O
  - Disk geometries not predictable
  - RAID: whole bunch of disks with data striped across them
    - Increases bandwidth, but does not change latency
    - Does not help small files (more on this later)

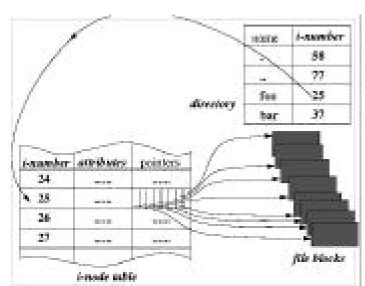
LFS Writes

- Treat disk as a tape!
- Buffer recent writes in memory
  - Log append only – no overwrite in place
  - Log is the only thing on disk! Main storage structure
- When you create a small file (less than a block):
  - Write data block to memory log
  - Write file inode to memory log
  - Write directory block to memory log
  - Write directory inode to memory log
- When memory accumulates to say 1MB or say 30 seconds have elapsed, write log to disk as a single write
- No seeks for writes
- But inodes are now floating

Floating I-nodes

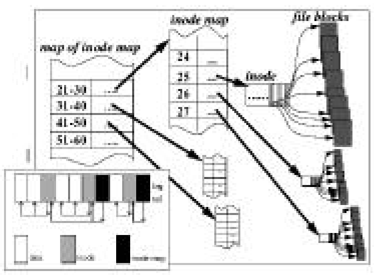
- Need to keep track of current position of inodes
- Requires an "I-node-map"
- I-node-map could be large (as many entries as there are files in the file system)
  - Break I-node-map into chunks and cache them
  - write out on the log those chunks that have changed
- Created a new problem!
  - How to find the chunks of I-node-map?
  - Create an I-node-map-map
- Have we solved the problem now?
  - I-node-map-map is small enough to be always cached in memory
  - It is small enough to be written to a fixed (and small position) on the disk (checkpoint region)
  - Write the I-node-map-map when filesystem is unmounted

Traditional Unix



- I-nodes stay fixed
- I-number translates to a disk location
- FFS splits this array but approach is similar

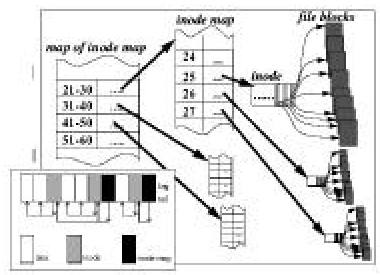
LFS: floating inodes



When write:

- Append data, inode, piece of inode-map to the log
- Record location of piece of inode map in map of inode map (in memory)
- Checkpoint map of inode map once in a while

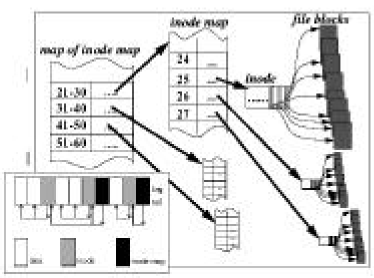
LFS Data structures



When read:

- From map map, to inode map, to inode to block
- Get some locality in inode map
- Cache a lot of hot pieces of inode map
- Number of I/Os per read: a little worse than FFS

LFS Data structures (contd.)



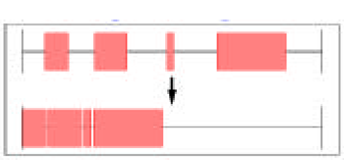
When recover:

- Read checkpoint, get map of map
- Roll forward in log to update map of map

Wrap Around Problem

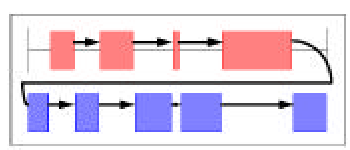
- Pretty soon you run out of space on the disk
- Log needs to wrap around
- Two approaches:
  - Compaction
  - Threading
- Sprite (first implementation of LFS):
  - Combination of the two; open up free segments & avoid copying

Compaction



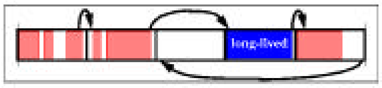
- Works fine if you have a mostly empty disk
- But suppose 90% utilization:
  - Write 10%
  - Compact: (read 90%, write 90%)
  - Creates 10% new free space
  - Spend 95% of time copying
- Should avoid compacting stuff that doesn't change

Threading



- Free space gets fragmented
- Pretty soon your runs start approaching minimum allocation size
- Same argument as not having large blocks and small fragments in FFS

Combined Solution



- Want benefits of both:
  - Compaction: big free space
  - Threading: leave long living things in place so they aren't copied again and again
- 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

Recap

- In LFS, everything is stored in a single log
  - Carry over the data-blocks and I-node data structures from Unix
  - Buffer writes and write them to disk as a sequential log
  - Use inode-map and inode-map-map to keep track of floating I-nodes
  - Cache (in memory) typically minimizes the cost of the extra levels of indirection
    - Inode-map-map and pieces of inode-map are cache in memory

Cleaning

- Eventually the log could fill the entire disk
  - Reclaim the holes in the log. Two approaches:
    - Compaction of entire disk
    - Threading over live data
  - LFS uses a hybrid strategy. Divides disk into "segments"
    - Threads over non-empty segments
    - Segments guarantee that seek costs are amortized
    - Every once in a while, picks a few segments, compacts them to generate empty segments

Cleaning Process

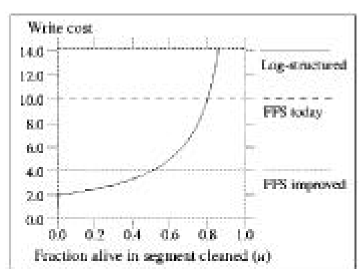
- When to clean?
  - When the number of free segments falls below a certain threshold
- Choosing a segment to clean:
  - Will be based on amount of live data it contains
  - Segment usage table: tracks number of live bytes in each segment
    - When you rewrite I-nodes/data blocks, find the old segment in which they used to live, and decrement the usage count for the old segment

Cleaning Process (contd.)

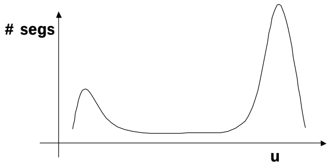
- How to clean?
  - Need to identify all of the live data in the segment
  - Segment summary block stores I-numbers (for I-nodes) and (I-number, block-number) for each data block
    - Check whether the corresponding data block still lives in that segment
    - Optimize this process by storing a version number with each I-number
      - when a file is deleted, increment this version number

Cleaning Cost

- Write cost = total_I/O / new_writes = (1+u+1-u)/(1-u) = 2/(1-u)
  - u better be small or it is going to hurt performance

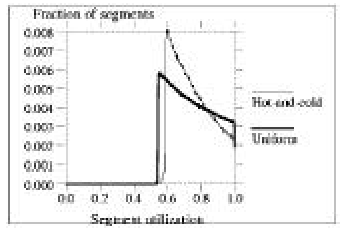


Cleaning Goals



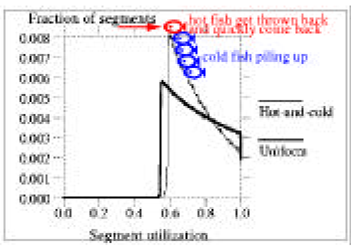
- Want bimodal distribution:
  - Small number of low-utilized segments
    - So that cleaner can always find easy segments to clean
  - Large number of highly-utilized segments
    - So that disk is well utilized

Greedy Cleaner



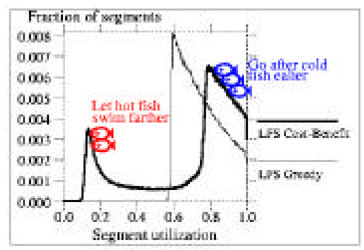
- Greedy cleaner: pick the lowest "u" to clean
- Workload #1: uniform (pick random files to overwrite)
- Workload #2: hot-cold workload (90% of the updates to 10% of the files)

Greedy Cleaner



- Greedy strategy is not creating a bimodal distribution
- Slow moving segments likely to make the cleaning threshold high
- Separation of data into hot & cold data also didn't help

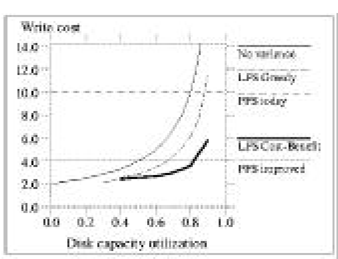
Better Approach



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

Cost-Benefit Analysis

- Optimize for benefit/cost = age*(1-u)/(1+u)
- Pick segments to clean based on highest "benefit/cost" value



Postscript

- Results:
  - 10x performance for small writes
  - Similar for large I/O performance
  - Terrible for sequential read after random writes
  - Fast recovery (support for transactional semantics)
- Then the fight started...
  - Margo Seltzer wrote Usenix papers that reported unfavorable performance of LFS
  - Resulted in a big controversial web warfare
  - Both sides made valid points. The debate was:
    - What's a representative workload?
    - How to draw the line between implementation artifacts and fundamental flaws of the approach?

When is LFS good?

- LFS does well on "common" cases
- LFS degrade for "corner" cases

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
- Provide insight with simplified math
- Simulation to evaluate and validate ideas
- Solid real implementation and measurements

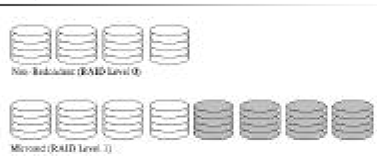
Announcements

- Design review meetings:
  - Tomorrow from 2-4pm
  - Thursday from 2-4pm with Zheng Ma
- Suggested background readings:
  - RAID paper
  - Unix Time Sharing System paper

RAIDs and availability

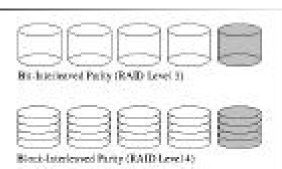
- Suppose you need to store more data than fits on a single disk (e.g., large database or file servers). How should arrange data across disks?
- Option 1: treat disks as huge pool of disk blocks
  - Disk1 has blocks 1, 2, …, N
  - Disk2 has blocks N+1, N+2, …, 2N
  - …………
- Option 2: Stripe data across disks, with k disks:
  - Disk1 has blocks 1, k+1, 2k+1, …
  - Disk2 has blocks 2, k+2, 2k+2, …
  - …………
- What are the advantages/disadvantages of the two options?

Array of Disks



- Storage system performance factors:
  - Throughput: number of requests satisfied per second
  - Single request metric: latency and bandwidth (could vary for reads and writes)
- RAID 0: improves throughput, does not affect latency
- RAID 1: duplicate writes; improves read performance (can choose closest copy, transfer large files at aggregate bandwidth of all disks)
  - Improves reliability (extra copy always available)

More RAID Levels



- No need for complete duplication to achieve reliability
- Use parity bits:
  - One scheme: interleave at the level of bits, store parity bit in parity disk
  - Another scheme: interleave at the level of blocks, store parity block in parity disk
    - Reads < block size: access only one disk (better throughput than RAID 3)

Writes to RAID 4

- Large writes which accesses all disks (say, a stripe of blocks)
  - Compute the parity block and store it on the parity disk
- Small writes. Two options:
  - Read current stripe of blocks, compute parity with the new block, write parity block
  - Better option:
    - Read current version of block being written
    - Read current version of parity block
    - Compute how parity would change:
      - If a bit on block changed, the corresponding parity bit needs to be flipped
    - Write new version of block
    - Write new version of parity block
- Disk containing parity block is updated on all writes

Distributed Parity

- Parity blocks are distributed across disks
  - Spreads load evenly
  - Multiple writes could potentially be serviced at the same time
  - All disks can be used for servicing reads

Comparison

- RAID-5 vs. normal disks:
  - RAID-5: better throughput, better reliability, good bandwidth for large reads, small waste of space
  - Normal disks: perform better for small writes
- RAID-1 vs. RAID-5: Which is better?
  - RAID-1 wastes more space
  - For small writes: RAID-1 is better
- HP-AutoRAID system:
  - Stores hot data in RAID-1
  - Cold data in RAID-5
  - Does automatic background propagation of data as working set changes