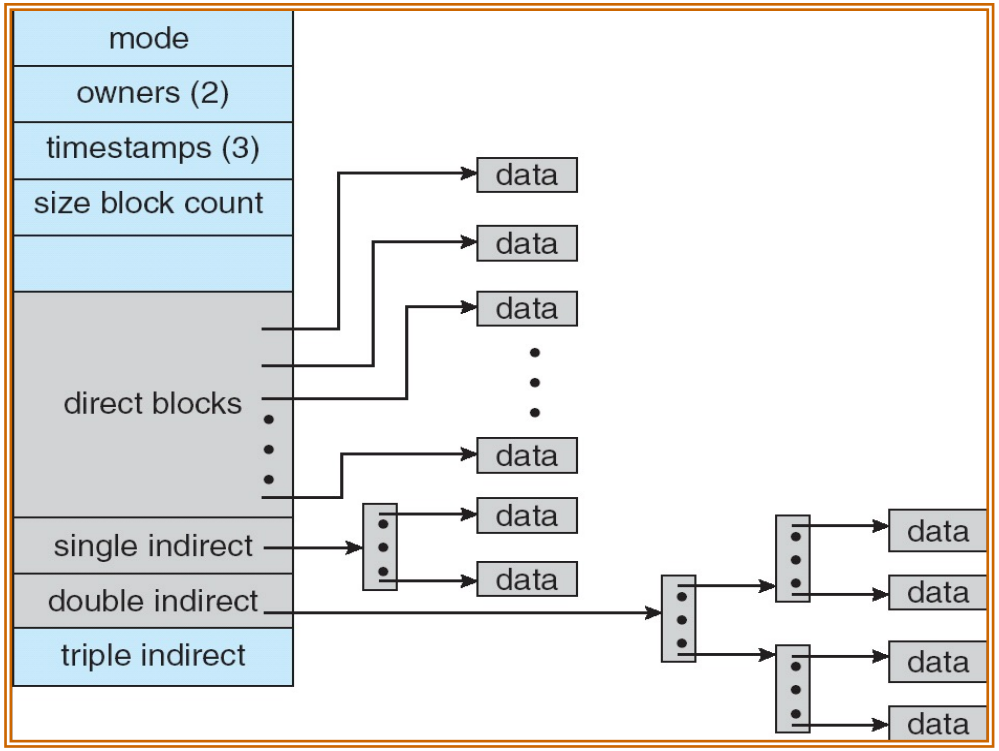
UNIVERSITY OF CALIFORNIA
UC RIVERSIDE

Filesystems (II)
FFS, LFS

Zooming in on i-node

- i-node: structure for per-file metadata (unique per file)
  - contains: ownership, permissions, timestamps, about 10 data-block pointers
  - i-nodes form an array, indexed by "i-number" – so each i-node has a unique i-number
  - Array is explicit for FFS, implicit for LFS (its i-node map is cache of i-nodes indexed by i-number)



- Indirect blocks:
  - i-node only holds a small number of data block pointers (direct pointers)
  - For larger files, i-node points to an indirect block containing 1024 4-byte entries in a 4K block
  - Each indirect block entry points to a data block
  - Can have multiple levels of indirect blocks for even larger files

Unix Inodes and Path Search

- Unix Inodes are not directories
- Inodes describe where on disk the blocks for a file are placed
  - Directories are files, so inodes also describe where the blocks for directories are placed on the disk
- Directory entries map file names to inodes
  - To open "/one", use Master Block to find inode for "/" on disk
  - Open "/", look for entry for "one"
  - This entry gives the disk block number for the inode for "one"
  - Read the inode for "one" into memory
  - The inode says where first data block is on disk
  - Read that block into memory to access the data in the file
- This is why we have *open* in addition to *read* and *write*

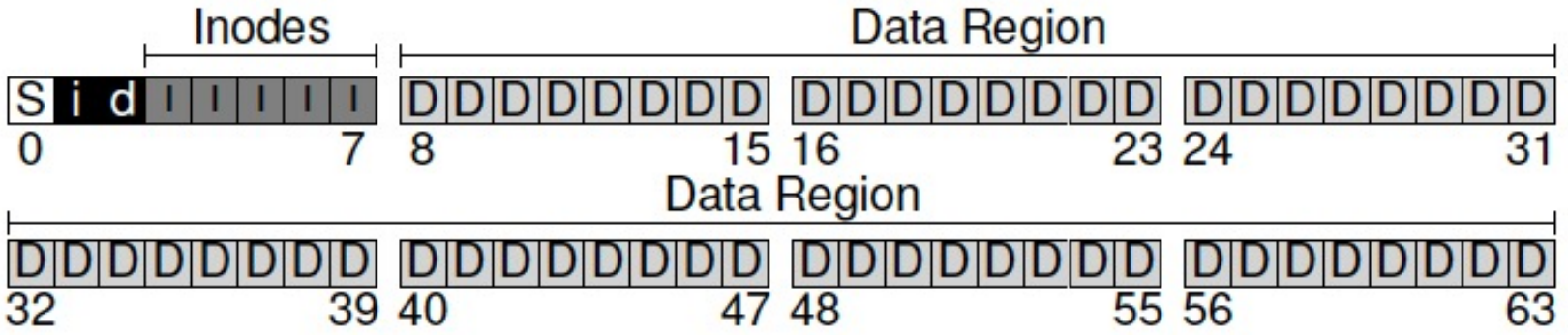
Every file/directory operation typically involves many I/O operations!

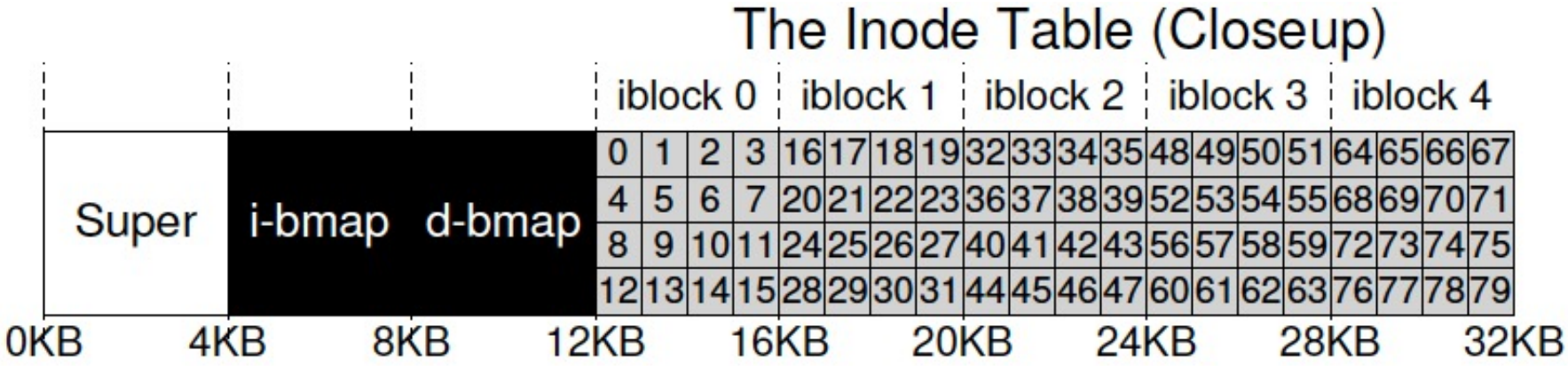
Caching and read ahead

UCR

- Unix file system implemented two additional optimizations
- Caching: keep a buffer cache in memory
  - Locality = avoid cost of expensive I/O
  - Even a relatively small cache makes a difference
- Read ahead: read the whole file (into cache) when it is opened
  - Many files accessed sequentially
  - Limited by cache size

A naïve implementation





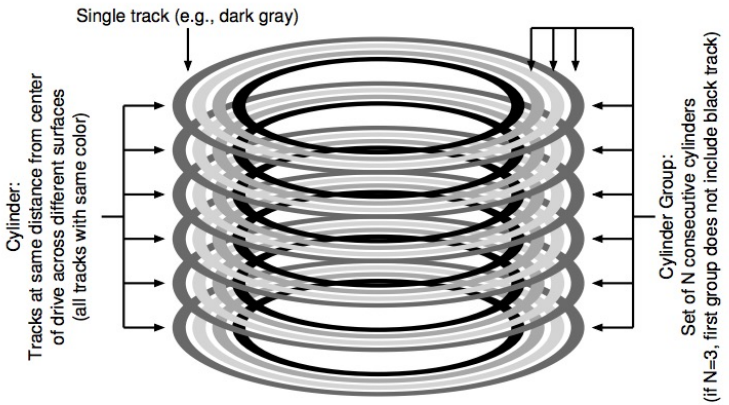
What's wrong with original unix FS?

UCR

- Original UNIX FS was simple and elegant, but slow
- Could only achieve about 20 KB/sec/arm; ~2% of 1982 disk bandwidth

- Problems:
  - Blocks too small
    - 512 bytes (matched sector size)
  - Consecutive blocks of files not close together
    - Yields random placement for mature file systems
  - i-nodes far from data
    - All i-nodes at the beginning of the disk, all data after that
  - i-nodes of directory not close together
  - no read-ahead
    - Useful when sequentially reading large sections of a file

FFS Changes -- Locality is important



- Aspects of new file system:
  - 4096 or 8192 byte block size (why not larger?)
  - large blocks and small fragments
  - disk divided into cylinder groups
  - each contains superblock, i-nodes, bitmap of free blocks, usage summary info
  - Note that i-nodes are now spread across the disk:
    - Keep i-node near file, i-nodes of a directory together (shared fate)
  - Cylinder groups ~ 16 cylinders, or 7.5 MB
  - Cylinder headers spread around so not all on one platter

FFS Locality Techniques

UCR

- Goals
  - Keep directory within a cylinder group, spread out different directories
  - Allocate runs of blocks within a cylinder group, every once in a while switch to a new cylinder group (jump at 1MB)
- Layout policy: global and local
  - Global policy allocates files & directories to cylinder groups – picks “optimal” next block for block allocation
  - Local allocation routines handle specific block requests – select from a sequence of alternative if need to

FFS Results

- 20-40% of disk bandwidth for large reads/writes
- 10-20x original UNIX speeds
- Size: 3800 lines of code vs. 2700 in old system
- 10% of total disk space unusable (except at 50% performance price)
- Could have done more; later versions do
- Watershed moment for OS designers– File system matters

FFS Summary

- 3 key features:
  - Parameterize FS implementation for the hardware it's running on
  - Measurement-driven design decisions
  - Locality "wins"
- Major flaws:
  - Measurements derived from a single installation
  - Ignored technology trends
- A lesson for the future: don't ignore underlying hardware characteristics
- Contrasting research approaches: improve what you've got vs. design something new

File operations still expensive

UCR

- How many operations (seeks) to create a new file?
  - New file, needs a new inode
  - But at least a block of data too
  - Check and update the inode and data bitmap (eventually have to be written to disk)
  - Not done yet – need to add it to the directory (update the directory inode and the directory data block – may need to split if its full)…
  - Whew!! How does all of this even work?

- So what is the advantage?
  - Not removing any operations
  - Seeks are just shorter…

Crash consistency/Journaling

- Problem:
  - Recall: file ops can cause multiple disk ops
    - Example, add a block to a file
  - With cache, and block reordering what happens on a crash?
    - Any subset of the ops may have been committed to disk
    - Disk state is inconsistent
      - File checkers where a critical part of the OS
- Solution: Journaling
  - Log intended ops in a journal – commit them
  - Once committed, start disk ops
  - Once committed, mark log as done
  - How does this solve the problem?

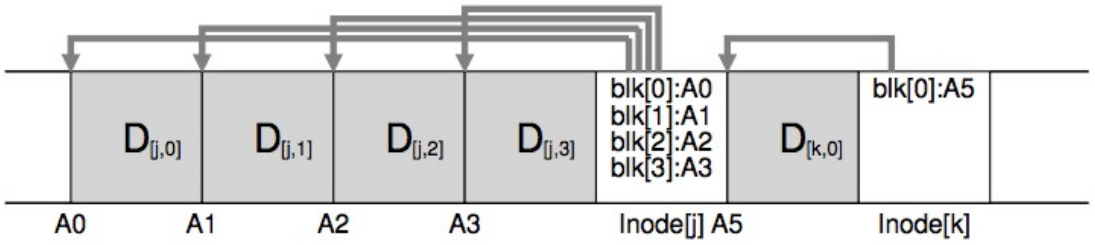
Log-Structured/Journaling File System

- Radically different file system design
- Technology motivations:
  - CPUs outpacing disks: I/O becoming more-and-more of a bottleneck
  - Large RAM: file caches work well, making most disk traffic writes
- Problems with (then) current file systems:
  - Lots of little writes
  - Synchronous: wait for disk in too many places – makes it hard to win much from RAIDs, too little concurrency
  - 5 seeks to create a new file: (rough order)
    1. file i-node (create)
    2. file data
    3. directory entry
    4. file i-node (finalize)
    5. directory i-node (modification time)
    6. (not to mention bitmap updates)

LFS Basic Idea

- Log all data and metadata with efficient, large, sequential writes
  - Do not update blocks in place – just write new versions in the log
- Treat the log as the truth, but keep an index on its contents
- Not necessarily good for reads, but trends help
  - Rely on a large memory to provide fast access through caching
- Data layout on disk has “temporal locality” (good for writing), rather than “logical locality” (good for reading)
  - Why is this a better? Because caching helps reads but not writes!

Basic idea

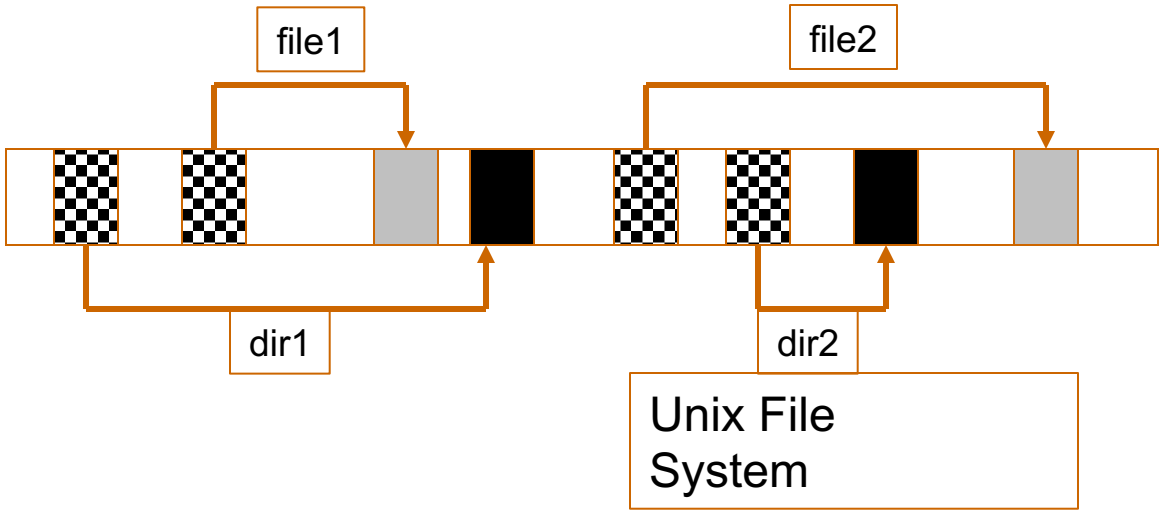


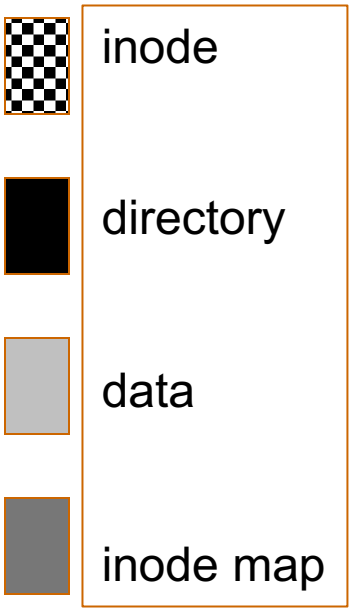
- We buffer all updates, and write them together in one big sequential write
  - Good for the disk
  - Example above, writes to two different files were written together (along with the new version of i-node) in one write
  - How much should we buffer?
    - What happens if too much? If too little?
- But how do we find a file??
  - All problems in CS solved with another level of indirection ☺

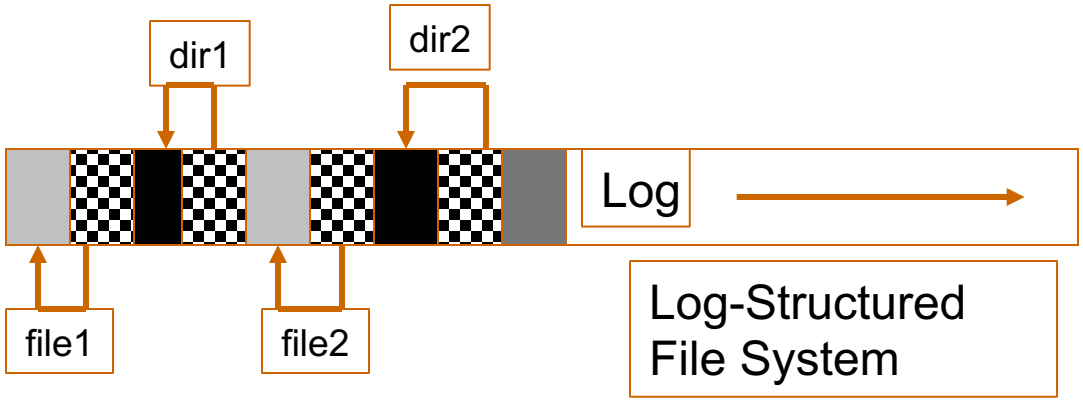
Devil is in the details

- Two potential problems:
  - Log retrieval on cache misses – how do we find the data?
  - Wrap-around: what happens when end of disk is reached?
    - No longer any big, empty runs available
    - How to prevent fragmentation?

LFS vs. UFS

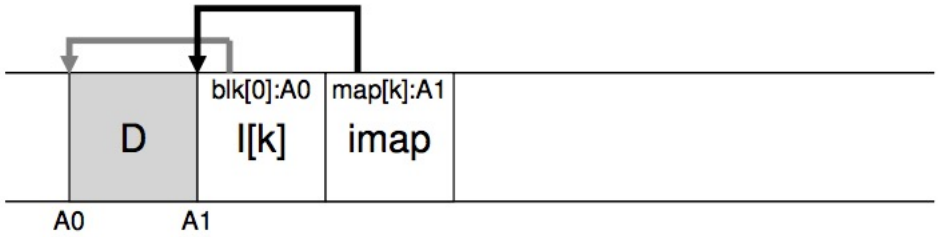






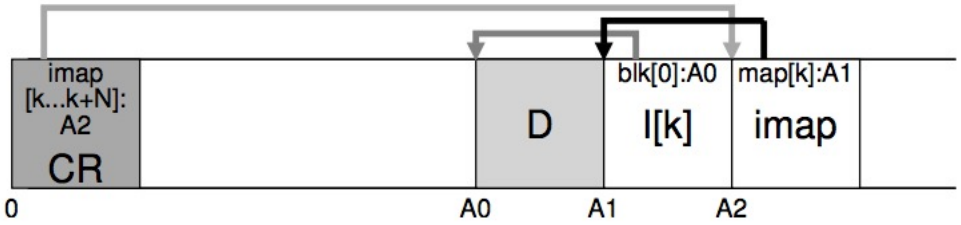
Blocks written to create two 1-block files: dir1/file1 and dir2/file2, in UFS and LFS

i-node map



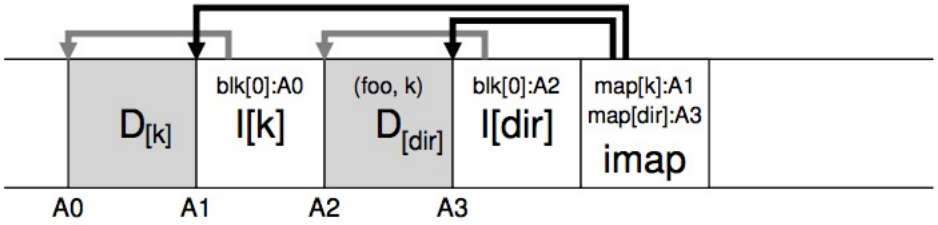
- A map keeping track of the location of i-nodes
- Anytime an i-node is written to disk, the imap is updated
  - But is that any better? In a second
- Most of the time the imap is in memory, so access is fast
- Updated imap is saved as part of the log!
  - but how do we find it!

Final piece to the solution



- Checkpoint region is written to point to the location of the imap
  - Also serves as an indicator of a stable point in the file system for crash recovery
- So, to read a file from LFS:
  - Read the CR, use it to read and cache the imap
  - After that, it is identical to FFS
  - Are reads fast?

What about directories?



- When a file is updated, its inode changes (new copy)
  - We need to update the directory inode (also creating a copy)
  - We need to update its parent directory
- Ugh….what to do?
  - Inode map helps with that too – just keep track of inode number and resolve it through inode map

LFS Disk Wrap-Around/Garbage collection

UCR

- Compact live info to open up large runs of free space
  - Problem: long-lived information gets copied over-and-over
- Thread log through free spaces
  - Problem: disk fragments, causing I/O to become inefficient again

- Solution: *segmented log*
  - Divide disk into large, fixed-size segments
  - Do compaction within a segment; thread between segments
  - When writing, use only clean segments (i.e. no live data)
  - Occasionally clean segments: read in several, write out live data in compacted form, leaving some fragments free
  - Try to collect long-lived info into segments that never need to be cleaned
  - Note there is not free list or bit map (as in FFS), only a list of clean segments

LFS Segment Cleaning

- Which segments to clean?
  - Keep estimate of free space in each segment to help find segments with lowest utilization
  - Always start by looking for segment with utilization=0, since those are trivial to clean…
  - If utilization of segments being cleaned is U:
    - write cost = (total bytes read & written)/(new data written) = 2/(1-U) (unless U is 0)
    - write cost increases as U increases: U = .9 => cost = 20!
    - Need a cost of less than 4 to 10; => U of less than .75 to .45
- How to clean a segment?
  - Segment summary block contains map of the segment
  - Must list every i-node and file block
  - For file blocks you need {i-number, block #}
  - Through i-map you check if this block is still being used for the (i-number, block #)

Evaluation Results

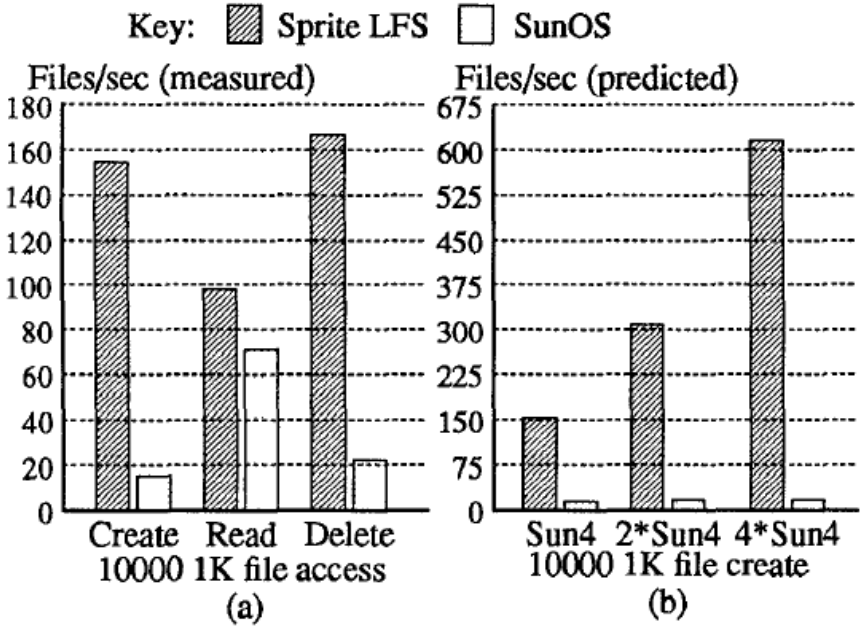


Figure 8 — Small-file performance under Sprite LFS and SunOS.
Figure (a) measures a benchmark that created 10000 one-kilobyte files, then read them back in the same order as created, then deleted them. Speed is measured by the number of files per second for each operation on the two file systems. The logging approach in Sprite LFS provides an order-of-magnitude speedup for creation and deletion. Figure (b) estimates the performance of each system for creating files on faster computers with the same disk. In SunOS the disk was 85% saturated in (a), so faster processors will not improve performance much. In Sprite LFS the disk was only 17% saturated in (a) while the CPU was 100% utilized; as a consequence I/O performance will scale with CPU speed.

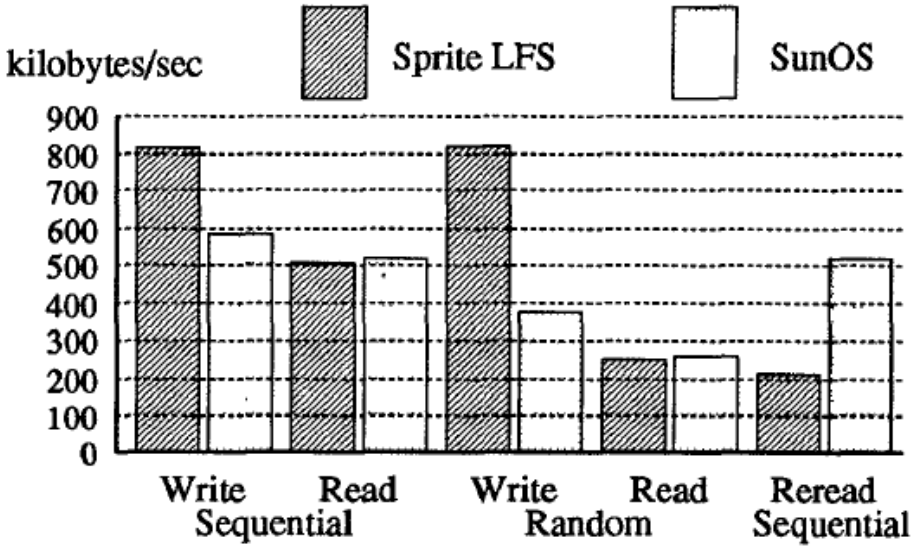


Figure 9 — Large-file performance under Sprite LFS and SunOS.
The figure shows the speed of a benchmark that creates a 100-Mbyte file with sequential writes, then reads the file back sequentially, then writes 100 Mbytes randomly to the existing file, then reads 100 Mbytes randomly from the file, and finally reads the file sequentially again. The bandwidth of each of the five phases is shown separately. Sprite LFS has a higher write bandwidth and the same read bandwidth as SunOS with the exception of sequential reading of a file that was written randomly.

Is this a good paper?

UCR

- What were the authors' goals?
- What about the evaluation/metrics?
- Did they convince you that this was a good system/approach?
- Does the system/approach meet the "Test of Time" challenge?
- How would you review this paper today?