The Zebra Striped Network File System

or, "Why you need a Log-Structured File System if you use RAID"

Presentation by
Ariel Tamches
6366 CS
tamches@cs.wisc.edu

Outline Of Talk

- Disk Arrays (RAID)
  - Performance potential and problems when using traditional file systems
  - conclusion: need to change underlying file system
- Log-structured File Systems (LFS)
  - Optimized to reduce disk seeks on writes
  - A perfect fit for RAID servers
- Zebra — RAID + LFS in a distributed file system
  - Components
  - Performance of Prototype

The CPU – I/O Performance Gap

Annual speedups:

CPU	50% or more
Disk access	10%
Disk transfer	20%

Amdahl's Law of Speedup:
(speed up only $frac_{enh}$ by factor of n)

$$
\begin{aligned}
speedup &= \frac{latency_{old}}{latency_{new}} \\
&= \frac{latency_{old}}{latency_{old}\{\frac{1}{n}frac_{enh} + (1 - frac_{enh})\}}
\end{aligned}
$$

Example:
Process is 60% CPU-bound; 40% I/O-bound
Increase CPU speed 5x (but not I/O), then
$speedup = 1.9$.

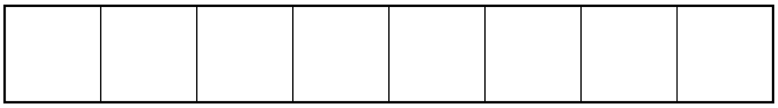
But disks *are* getting cheaper; can we make use of that?

RAID – disk arrays

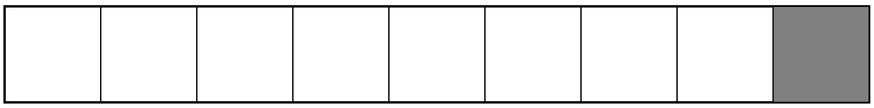
Array of disks which act like one faster disk.
Files are *striped* across n disks
The n portions are accessed in parallel

Start with:

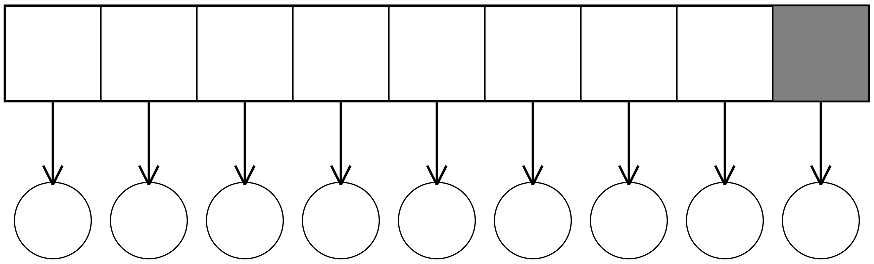
Chop into n striping units:



Add parity:



Send to RAID disks in parallel:



Reads are analagous (reverse the steps).

The Performance Potential of RAID

For an n disk RAID, bandwidth is the sum total:

$$bandwidth_{new} = \sum_{i=1}^{n} bandwidth_i$$

- Transfer do indeed work in parallel
- Parity can be computed and written on the fly with little slowdown.

But...

Problems with RAID

- What if a file doesn't fill up a stripe?
  - need to choose a "small write" policy

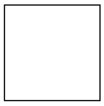
- Seek and Rotational Latency are not improved

- Goals of increasing parallelism and reducing disk seeks are in conflict
  - tough to utilize all n disks efficiently if n is high

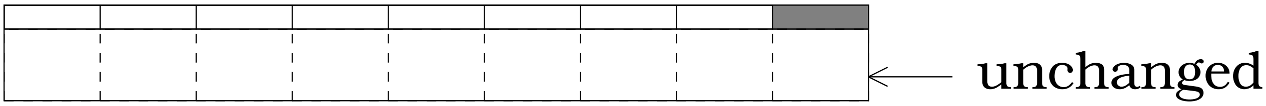
Small File Choices

What to do when file size < stripe size?
Two possibilities:

- "Thin RAID" (RAID level 3)

data = 

chop into n equal units, each < striping unit:



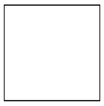
+ Easy to implement

+ Utilizes all disks

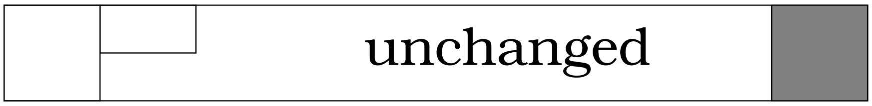
— Fragments the disks

Small File Choices, cont'd

- "Non-RAID" (RAID levels 4, 5)

data = 

fill $k < n$ striping units:



+ Does not fragment the disks

— Only $k < n$ disks are used (and usually $k \ll n$)

— Parity must be updated with
read-modify-write series

- Neither alternative is attractive
- Bad news: most accesses are to small files in Unix

$\Rightarrow$ 42% to $\leq$ 1K files
$\Rightarrow$ 80% to $\leq$ 10K files

Seek and Rotational Latencies in RAID

Disk transfer time is only one part of total transfer time.

$$\text{total time} = \underbrace{\text{seek time}}_{\sim 11\text{ms}} + \underbrace{\text{rotational latency}}_{\sim 5.5\text{ms}} + \underbrace{\text{transfer time}}_{\text{file size / (4.6 MB/sec)}}$$

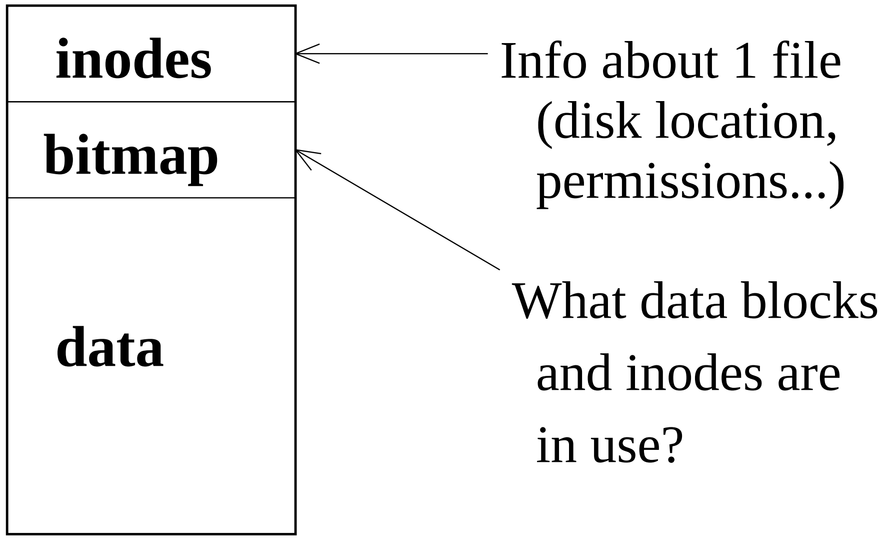
Only transfer time (the third component) is improved!
RAID will do well only when transfer time dominates.

What can we expect in practice?

Seek and Rotational Latencies, Cont'd

Underlying file system affects performance

Unix FFS: split disk into cylinder groups of files with locality



- To read a file

access file's inode; read inode; access file; read file

- To write a file (assume cached bitmap)

access directory inode and read;
access directory and read;
access file inode and write;
access file and write

Seek and Rotational Latencies, Cont'd

- Realistic Test—writing a 10K file
- Assume a cache with buffered write-through

Without RAID:

$$\begin{aligned}
latency_{old} &= seek_{dir-inode} + rot\ latency_{dir-inode} + \\
&\quad read_{dir-inode} \\
&+ seek_{dir} + rot\ latency_{dir} + write_{dir} \\
&+ seek_{inode-file} + rot\ latency_{inode-file} + \\
&\quad write_{inode-file} \\
&+ seek_{file} + rot\ latency_{file} + write_{file} \\
&= 4(11ms + 5.5ms) + \frac{1+1+1+10K}{4.6MB/sec} \\
&= 66ms + 2.8\ ms
\end{aligned}$$

With 10-disk RAID:

$$latency_{new} = 66ms + \frac{1}{10}2.8\ ms$$

A speedup of only 4%! (not 1000%)

Even with a disk cache, this is realistic because we need to write-through for safety.

Balancing Disk Load

The goals of

- Balancing load (using all n disks) and
- Reducing disk seeks

tend to conflict!

Why? Using all disks for each operation requires "thin-RAID" strategy for small files, leading to fragmentation and hence increased high ratio of seek time to transfer time.

$\Rightarrow$ The more we scale n, the more "thin-RAID" files we get.

Can we have scalable (large n) RAIDs that don't lead to "thin-RAID" files in practice? *Not without major change to underlying file system*

Good RAID performance requires a file system that

- Reduces disk seeks
- Fills up the stripes

Log-structured File Systems (LFS)

A radical departure from Unix FFS

Carefully optimized to reduce/eliminate disk seeks in the presence of a write-through disk cache.

Why assume *write-through* cache?
For safety (write buffering still encouraged)

$\Rightarrow$ So, disk accesses will be dominated by "write-throughs"

We will see that LFS and RAID are a perfect match

LFS — How it works

All file writes—files, inodes, and directories—are appended to the “log”, the disk’s only structure.

This avoids disk seeks on writes, but:

- How do reads work, now that the inode disk address is not fixed?
- We will eventually run out of disk space if we keep appending.

Reads in LFS

Now, the inode is appended to the log just like files.
An "inode map" (filenames $\rightarrow$ inode locations)
is provided at a fixed location in the log.

In practice, most of the inode map can be cached,
analogous to a page table.

So reads can work, but they are *no faster* in LFS
than in Unix FFS. Seeks are still needed.
$\Rightarrow$ But assuming an effective disk cache,
there are few disk reads.

Cleaning the Log

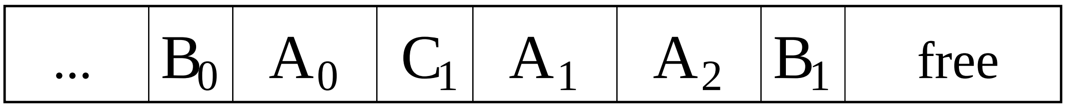
If we keep appending changes to the log, we will obviously run out of disk space.

- Need a "cleaning policy" to purge outdated files (whether deleted or updated)

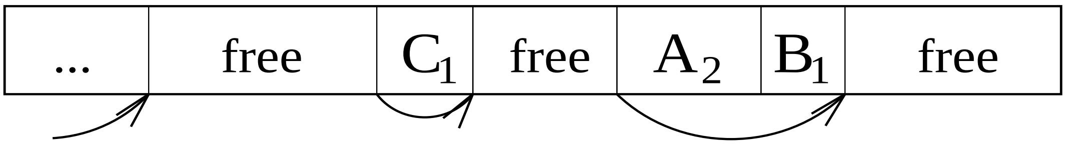
⇒ This is the toughest challenge of an LFS
⇒ A naive choice can fragment the disk

Cleaning Policy—Threading

Assume cleaner is run on the log



Threading creates a linked list of free space, jumping around still-valid files



+ Cleaning is quick

— Creates a fragmentation problem for future writes

— Will be too many disk seeks in the future

Not a good choice

Cleaning Policy—Copying

Assume cleaner is run on the log

...	B_0	A_0	C_1	A_1	A_2	B_1	free

Copying moves the still-valid data to a new contiguous space in the log. Free space is unthreaded.

...	free	C_1	A_2	B_1	free

+ Leaves contiguous free space for future writes

— Cleaning is too slow

Also not a good choice.

However, a *combination* of threading and copying works well.

Cleaning Policy—Segments

Divide the disk into large (about 1 MB) "segments"
The granularity of cleaning is by-segment, not by-file.

- If a segment gets cleaned, its live data are copied out contiguously into a new segment
- Threading on granularity of segments is used
- At worst, seeks every 1MB while writing
  A good compromise

Once segment(s) to clean are chosen, cleaning is easy:

- Read n segments into memory
- Identify live data
- Write $m < n$ new segments containing the live data
- The original n segments are now available

How do we know which data in a segment are live?
Which segments should we choose to clean?

LFS Cleaning—Identifying Live Data

A "segment summary block" is written as data enters the segment. It contains sufficient information for identifying live blocks.

Need only to point to inode of owning file. By reading the inode and checking whether it points to this block, we know if the block is live.

In practice, optimizations are used

LFS Cleaning—Choosing Segments

$\Rightarrow$ Most important performance parameter in an LFS

- Want to clean efficiently so we don't have to do it again soon
- Clean segments with little or no live data to make copying cheap

But, long-lived (cold) and short-lived (hot) segments should also be accounted for.

- Cleaning a hot segment is not useful; the newly written segment's files will die quickly and we'll have to clean it again soon
- Cleaning a cold segment is useful; free space created by cleaning it will last.

RAID and LFS together

Recall RAID needs file system that

- Reduces seeks
- Writes no small files, or better yet: writes entire stripes at a time

Both goals are achieved in an LFS.

Seeks are needed only:

- For Cleaning
- For Handling cache read misses
- Once per MB during writing (at worst) due to threading

$\Rightarrow$ Rig RAID stripe size $\equiv$ write buffer size $\equiv$ segment size

- Now, we can usually fill the RAID stripes perfectly!
- No more small write problem

Zebra Network Filing System

- Striping
- Components
- Algorithms
- Performance of Prototype

Zebra Striping

Recall vanilla RAID uses "per-file" striping;
each file striped on its own.

- "Thin-RAID" policy:

— Fragments the disks

- "Non-RAID" policy:

— Only $k < n$ disks are used (and usually $k \ll n$)

— Parity updates slower

Zebra Striping, cont'd

The granularity of striping in Zebra is by-log.

- Each user in the network file system has an LFS log
- Assuming write buffers, logs are written and parity is calculated on granularity of 1MB segments

+ Stripes are always filled; all n server disks are used
+ Seeking greatly reduced
+ Small file writes are fast
+ Large file writes are fast
+ Parity writing simplified
+ Scalable performance—increase n almost at will

Zebra Components

- Clients
  - Each client maintains its own log; writes go through to storage servers
- Storage Servers—the RAID servers
  - Stores client logs—but only file data; not inodes, directories, or other “miscellanea”
- File Manager
  - No file data stored here—just “miscellanea”
  - Protection and Security enforced here—enables secure file sharing
  - Clients get inodes, etc. from file manager and then go to storage servers to read actual file
  - Implemented as an LFS itself (non-RAID)
- Stripe Cleaner
  - Reclaims space on storage servers

Zebra Algorithms

- Writing
- Reading
- Cache Consistency

Zebra Writing

Write-thru caches (with write buffers) exists on each client
Cache is written to storage server when:

- cache is full, or
- 30 seconds elapse, or
- explicit `fsync` from application
- file manager requests it to maintain cache consistency

Most files (65-80%) have lifetimes under 30 seconds

- Due to buffering, may *never* have to write such files!

Writes are asynchronous.

Transaction Processing programs perform poorly

- Very frequent `fsync's`
- LFS no better than Unix FFS

Zebra Reading

As discussed earlier, a "non-issue" assuming effective caching.

But what about *huge* files (e.g. $1024 \times 1024 \times 24 \times t$ byte animations)?

- Prefetching becomes an important topic

Zebra Cache Consistency

Zebra is built on top of Sprite, which provides exact Unix semantics (multiple distributed writers permitted with perfect consistency maintained).

Sprite (and hence Zebra) have a major performance flaw:

- File data cached just fine, but
- Naming is not cached; client must go to server on all file opens to perform expensive pathname→inode translation

Zebra: will do name caching in the future.
However, would still need to go to server on each open to ask whether a cached file is consistent.
⇒ Zebra is not a satisfactory distributed file system, as AFS is. (Neither are NFS, Locus, Sprite)
Zebra should consider sacrificing "exact Unix semantics".

Performance of Prototype

- Reminder: Zebra did not use name caching
  - So small access will be very slow

Zebra Configuration: 5 disks (4 in RAID storage manager), 1 for the file manager.

Large file write throughput:

- Zebra 3 clients/5 servers: 3.25 MB/sec
- Zebra 1 client/5 servers: 2.6 MB/sec
- Sprite (1 client, 1 server): 0.5 MB/sec
- NFS (1 client, 1 server): 0.5 MB/sec

Large file read throughput:

- Zebra 3 clients/5 servers: 5.1 MB/sec
- Zebra 1 clients/5 servers: 2.8 MB/sec
- Sprite (1 client, 1 server): 0.5 MB/sec
- NFS (1 client, 1 server): 0.5 MB/sec

Performance of Prototype cont'd

Small file write benchmark response time (1 client):

- NFS: 62 seconds
- Sprite LFS (no name caching): 21 seconds
- Zebra (no name caching): 17 seconds
- Zebra (name caching estimate): 6 seconds

- The Best News Of All...

For the small file benchmarks, Zebra kept sum of client and file manager CPUs (25 MHz) near 100% (could not be more due to synchronous RPC).

- Just pop in enhanced CPU

Will the file manager become a bottleneck?

- File manager disk was from 0 to 5% utilized;
  file manager CPU was > 95% idle for large benchmark, and 55% idle for small benchmark

⇒ 55% value will go up once proper file name caching is implemented.

We didn’t cover...

- RAID crash recovery
- LFS crash recovery
- Performance of cleaning
- Is access permitted during cleaning?

For More Information...

Measurements of Realistic file systems:

- Baker, Hartman, Kupfer, Shirriff, and Ousterhout. "Measurements of a Distributed File System". SOSP 13; October, 1991

RAID:

- Chen, Lee, Gibson, Katz, and Patterson. "RAID: High-Performance, Reliable Secondary Storage". ACM Computing Surveys; June, 1994

LFS:

- Rosenblum and Ousterhout. "The Design and Implementation of a Log-Structured File System". ACM TOCS; February, 1992

Zebra:

- Hartman and Ousterhout. "The Zebra Network Striped File System". SOSP 14; December, 1993