File System Performance
CS239
Advanced Operating Systems
April 21, 1997

Outline

- Basics of file system performance
- Methods of improving file system performance
- Evolution in file system performance improvements

File System Performance Issues

Much computing activity makes heavy use of the file system

E.g., Web servers keep pages in files

For many applications, file system performance is key to system performance

File system performance heavily dependent on storage technology

Disk Storage Systems

Most on-line mass storage today is disk

Disk-based file system performance limited by -

- Disk seek
- Rotational latency
- Disk access time
- Disk bandwidth

Typical Disk Overheads

9-15 msec seek time

Apx. 8 msec rotational delay

Apx. .25 msec to transfer 1000 byte block

Apx. 4.5 Mbyte per second BW

So 17-23 msec to get 1000 bytes at random from disk

Some Disk File System Basics

Files are typically stored in one or more blocks of a disk

File metadata is usually stored separately from file data

Files must be preserved from corruption when crashes occur

Other Typical Disk Characteristics

From HPUX study, Usenix 1993
57% of disk accesses are writes
18-33% of reads sequential
8-12% of writes sequential
50-75% of all I/Os are synchronous
67-78% of writes are to metadata
13-42% of total disk access for user I/O
10-18% of all writes are to last written block

Implications of These Numbers

- *57% of disk accesses are writes*

Optimizing write performance is a very good idea

- *18-33% of reads sequential*

Read-ahead of blocks likely to win

- *8-12% of writes sequential*

Perhaps not worthwhile to focus on optimizing sequential writes

More Implications

- *50-75% of all I/Os are synchronous*

We're paying a big cost to keep file systems consistent

- *67-78% of writes are to metadata*

Optimizing metadata writes very helpful

- *13-42% of total disk access for user I/O*

Purely concentrating on user patterns won't solve the problem

Even More Implications

- *10-18% of all writes are to last written block*

Some savings possible by clever delay of writes

Remember, though - **these figures are specific to one file system!**

File System Performance Parameters

- Read latency
- Write latency
- Bandwidth delivered

Basic File System Performance Improvement Methods

- Increase locality
  - To minimize seek latency
  - Improves latency times
- Improve data layout
  - To make larger transfers possible
  - Improves bandwidth (and sometimes latency)
- Satisfy requests from main memory
- Use more than one disk

Typical Disk Overheads

9-15 msec seek time

Apx. 8 msec rotational delay

Apx. .25 msec to transfer 1000 byte block

Apx. 4.5 Mbyte per second BW

So 17-23 msec to get 1000 bytes at random from disk

Poor Bandwidth Usage Example

Using disk figures from previous slide

Getting to start of sector takes ~ 20 msec

If sector 1K long, ~ .25 msec to read

Total time to read 1000 bytes - ~21 msec

Effective BW is only 50,000 bytes/sec

Unix Fast File System utilizes only 5-10% of disk bandwidth

How Are Disks Improving?

- Density measurements improve 10-25% per year
- Capacity improves 25% per year
- Transfer rate improves 20% per year
- Seek time improves 8% per year
- Rotational latency improves 5-8% per year

All slower than processor speed increases

The Disk/Processor Gap

Since processor speeds double every two to three years

And disk seek times double every ten years

Processors are waiting longer and longer for data from disk

Important for OS to cover this gap

What Can the OS Do?

- Minimize amount of disk access
- Improve locality on disk
- Maximize size of data transfers
- Fetch from multiple disks in parallel

Minimizing Disk Access

Avoid disk accesses when possible

Use caching to hold file blocks in memory

Buffer writes

– Effect - decreases latency by removing the relatively slow disk from the path

Caching

OS controls I/O buffer cache

Typically with LRU methods

Generally used for all I/O, not just disk

What size should the buffer cache be?

Varies, but 10% of total system memory is common

Buffer Cache Design Factors

- Most files are short (but long files can be very long)
- User access is bursty
- File accesses are largely sequential
  - Up to 70-90% of all accesses
- Most files aren't open long
  - Up to 75% for < 1/4 second
- File lifetimes are short
  - Up to 65-80% live < 30 seconds

What Does This Imply About Buffer Caches?

- Design for holding small files
- Read-ahead is good
  - When access is sequential
- Dynamically changing the size of the cache might help

Read-Ahead

OS attempts to anticipate disk needs of programs

By reading blocks they are likely to request later

During time disk would otherwise be idle

Pros and Cons of Read-Ahead

+ Very good for sequential access of large files

  E.g., executables

+ Allows immediate satisfaction of disk requests

– Reduces number of LRU blocks in cache

– Extra OS complexity

Buffering Writes

Buffer writes so that they needn't be performed at once

Reducing latency on writes

But buffered writes are asynchronous

Potential cache consistency and crash problems

Some systems make certain critical writes synchronously

Should We Buffer Writes?

If files don’t live long, buffered writes might be good

- But danger of losing data in face of crashes
- And most short-lived files are also short in length
- 1/4 of all bytes deleted/overwritten in 30 sec.

Additional hardware could help

Improving Locality

Make sure next disk block you need is close to the last one you got

File layout is important here

Ordering of accesses in controller helps

– Effect - less seek time & rotational delay

Maximizing Data Transfers

Transfer big blocks or multiple blocks in one read

Tradeoff against wasted disk space

Readahead is one good method here

– Effect - Increase effective disk bandwidth and reduce number of disk I/Os

Fetching From Multiple Disks in Parallel

Multiprogramming can cause some of this automatically

Use of disk arrays can parallelize even a single process’ access

- At the cost of extra complexity

– Effect - Increase effective disk bandwidth

Evolution in File System Performance Work

- Unix Fast File System
- Log structured file system
- RAID
- Zebra
- Serverless file systems

Unix Fast File System

Designed to improve performance of Unix file I/O

Two major areas of performance improvement -

- –Bigger block sizes
- –Better on-disk layout of files

Block Size Improvements in FFS

Quadrupling of block size quadrupled amount of data gotten per disk fetch

But could lead to fragmentation problems

So block fragments introduced

- Small files stored in fragments
- Fragments addressable (but not independently fetchable)

Disk Layout Improvements

Aimed towards avoiding disk seeks

Bad if finding related files takes many seeks

Very bad if finding all the blocks of a single file requires seeks

FFS tried to keep these related things close together on disk

Cylinder Groups

A cylinder group is a set of consecutive disk cylinders in the FFS

FFS tries to keep files in the same directory in the same cylinder group

Within cylinder group, tries to keep "contiguous"

But must not let a cylinder group fill up

- So new directories treated specially

Locations for New Directories

Put new directory in relatively empty cylinder group

What is "empty"?

- Many free inodes
- Few directories already there

Avoids clustering all files in one cylinder group

The Importance of Free Space

File systems must not run too close to disk capacity

No room for new files

Layout policies ineffective when too few free blocks

Typically, FFS needs 10% of the total blocks free to perform well

Performance of FFS

4 to 15 times the bandwidth of old Unix file system

Depending on size of disk blocks

Performance on original file system limited by CPU speed

- Due to memory-to-memory buffer copies

Unix FFS is Not the Ultimate Solution

Based on technology of the early 80s

And file usage patterns of those times

In modern systems, FFS achieves only 5-10% of raw disk bandwidth

Where Did FFS Look For Performance Improvements?

- Increased size of data transfers
  - Bigger blocks
- Improving locality
  - Locating file's blocks and blocks of related files close together

The Log-Structured File System

Good, large buffer caches can catch almost all reads

But most writes have to go to disk

So file system performance can be limited by writes

So, produce FS that writes quickly

Like an append-only log

Basic LFS Architecture

Buffer writes, send them sequentially to disk

- Data blocks
- Attributes
- Directories
- And almost everything else

Converts small sync writes to large async writes

A Simple Log Disk Structure

File A Block 7	File Z Block 1	File M Block 202	File A Block 3	File F Block 1	File A Block 7	File L Block 26	File L Block 25

Head of Log →

Key Issues in Log-Based Architecture

1. Retrieving information from the log

   No matter how well you cache, sooner or later you have to read

2. Managing free space on the disk

   You need contiguous space to write - in the long run, how do you get more?

Finding Data in the Log

File A	File Z	File M	File A	File F	File A	File L	File L
Block 7	Block 1	Block 202	Block 3	Block 1	Block 7	Block 26	Block 25

Give me block 25 of file L
Or,
Give me block 1 of file F

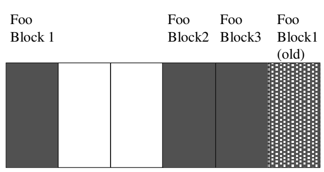
Retrieving Information From the Log

Must avoid sequential scans of disk to read files

Solution - store index structures in log

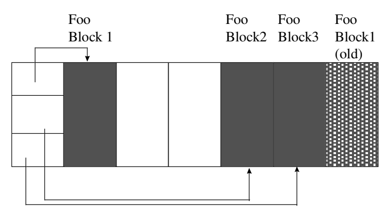
Index is essentially the most recent version of the inode

Finding Data in the Log



How do you find all blocks of file Foo?

Finding Data in the Log With an Inode



How Do You Find a File’s Inode?

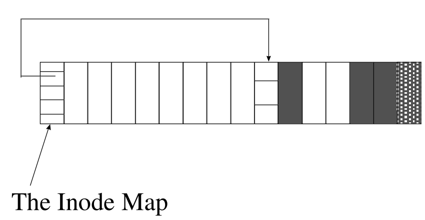
You could search sequentially

LFS optimizes by writing *inode maps* to the log

Inode map points to the most recent version of each inode

A file system’s inodes cover multiple blocks of inode map

How Do You Find the Inode?



How Do You Find Inode Maps?

Use a fixed region on the disk

That always points to the most recent inode map blocks

But cache inode maps in main memory

Small enough that few disk accesses required to find inode maps

Reclaiming Space in the Log

Eventually, the log reaches the end of the disk partition

So LFS must reuse disk space

Like superseded data blocks

Space can be reclaimed in background

Or when needed

Goal is to maintain large free extents on disk

LFS Space Reclamation Strategy

Combination of copying and threading

Copy to free large fixed-size *segments*

Thread free segments together

Try to collect long-lived data permanently into segments

Run cleaner in background periodically

Choosing which segments to clean is tricky

Performance of LFS

On modified Andrew benchmark, 20% faster than FFS

LFS can create and delete 8 times as many files per second as FFS

LFS can read 1 1/2 times as many small files

LFS slower than FFS at sequential reads of randomly written files

Where Did LFS Look For Performance Improvements?

- Minimized disk access
  - Only write when segments filled up
- Increased size of data transfers
  - Write whole segments at a time
- Improving locality
  - Assuming temporal locality, a file's blocks are all adjacent on disk
  - And temporally related files are nearby

Parallel Disk Access and RAID

One disk can only deliver data at its maximum rate

So to get more data faster, get it from multiple disks simultaneously

Saving on rotational latency and seek time

Utilizing Disk Access Parallelism

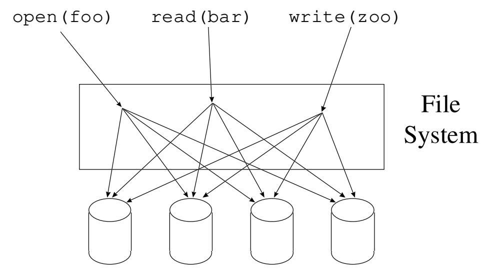
Some parallelism available just from having several disks

But not much

Instead of satisfying each access from one disk, use multiple disks for each access

Store part of each data block on several disks

Disk Parallelism Example



Data Striping

Transparently distributing data over multiple disks

Benefits -

- Increases disk parallelism
- Faster response for big requests

Major parameters are number of disks and size of data interleaf

Fine-Grained Vs. Course-Grained Data Interleaving

- Fine grain data interleaving
  - +High data rate for all requests
  - –But only one request per disk array
  - –Lots of time spent positioning
- Course grain data interleaving
  - +Large requests access many disks
  - +Many small requests handled at once
  - –Small I/O requests access few disks

RAID Architectures

Redundant **A**rrays of **I**ndependent **D**isks

Basic architectures for organizing disks into arrays

Assuming independent control of each disk

Standard classification scheme divides architectures into levels

Block-Interleaved Distributed-Parity (RAID Level 5)

Sort of the most general level of RAID

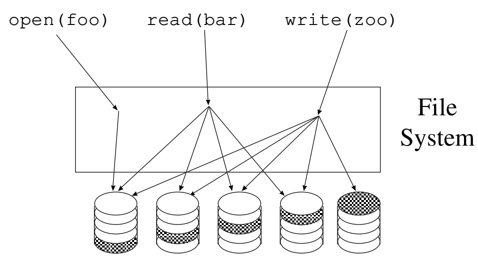
Spread the parity out over all disks

\+ No parity disk bottleneck

\+ All disks contribute read bandwidth

– Requires 4 I/Os for small writes

Block-Interleaved Distributed-Parity Diagram



Where Did RAID Look For Performance Improvements?

- Parallel use of disks
  - Improve overall delivered bandwidth by getting data from multiple disks
- Biggest problem is small write performance
- But we know how to deal with small writes . . .

Zebra

Many machines today use remote file access

Files are often gotten from a server over a high speed LAN

Conceptually, those servers are like disks and the LAN is like a bus

But we don't get data at the "bus" speed

Performance Problems With Remote Servers

- They still have to get the files off disks
  - Network transmission time often better than local disk fetch
  - But remote disk fetch plus network transmission really stinks
- If too many people use a server, it becomes a bottleneck

Caching for Remote Data Access

Caching helps a lot

Multi-level caching helps more

Cache at the client machine

Also cache at the server machine

Cooperative caching helps even more

Use the caches of other machines before going to disk

Applying Parallelism To Remote File Access

If remote servers are disks, why not turn several of them into a "RAID"?

Spread writes across multiple cooperating remote file servers

Reads handled the same way

Potential Problems With Parallelizing File Servers

- Machines crash more often than disks
- Small writes are terrible

The Zebra Striping File System

Stripe writes across multiple servers

Use RAID-style redundancy to handle failures

Use log structured file system techniques to avoid ever performing small writes

Important Problems in Zebra

- How do I find my files?
  - More precisely, how does the system find them?
- Log segment cleaning is now a distributed process
- Cache consistency

Finding Files

Ask the file manager

Single machine that stores all the file metadata

Caching used to avoid bottlenecks

Metadata stored using Zebra itself

So crashed file manager can be replaced easily

Log Segment Cleaning

Unlike log-structured file system, not everything to be cleaned is on one machine

Stripe cleaner process runs periodically

Gathers data from participating machines and issues cleaning instructions

Cache Consistency in Zebra

If more than one client caches Zebra data, how do we assure consistency?

Centralized at the file manager

Another potential bottleneck

Where Did Zebra Look For Performance Improvements?

- Parallel use of disks and file servers
  - Improve overall delivered bandwidth
- Small write performance improved through log structure
- File system services (metadata handling, cache coherence, stripe cleaning) are a potential bottleneck