Optimisations in File Systems

Stephen Tweedie

Outline

- Goals of FS Optimisations
- Optimisation Techniques
- Data Allocation
- Reliability
- Fragmentation
- Conclusion

Goals of FS Optimisation

- Bandwidth: maximize raw data throughput to and from disk
- Latency: rapid response time to I/O requests
- Efficiency: low CPU and storage overheads
- Fairness: even distribution of disk resources
- Load: performances acceptable under multiple concurrent I/O requests
- Reliability: resistance to, and recovery from, crashes

Optimisations Techniques

- Buffer Cache and I/O Scheduling
- Device Drivers Improvements
- I/O Ordering

Buffer Cache and I/O Scheduling

- Unified buffer cache:
  - Buffer space is shared with process address space when possible
  - Memory efficiency means that large buffer caches can be supported
- I/O scheduling:
  - Writes are usually asynchronous (except writes to the swap areas)
  - Reads are synchronous and, thus, time-criticals
  - Prioritize I/O
  - Reorder data requests to maximize throughput

Device Drivers Improvements

- IDE multi-buffer code
- SCSI scatter-gather: allows efficient access to adjacent disk blocks in non-adjacent memory regions
- Clustering:
  - improves over scatter-gather by trying to bring adjacent buffers into physical adjacent memory
  - works for non-SCSI disks too

I/O Ordering (Read-Ahead)

- BSD FFS:
  - one block readahead
  - works well unless multiple processes are active (the head trashes)
- Viva FS: algorithmic lookahead depending upon the current load and file access history
- Linux Ext2fs:
  - performs readaheads for sequential reads
  - extends readahead to explicit (readdir(2)) and implicit (namei kernel directory lookup) directory reads
  - number of blocks to read-ahead is device specific

Data Allocation

- System V Release 3
- BSD FFS
- Viva FS
- Ext2 FS

Data Allocation in SVR3

- Derived from Unix V7 FS
- No special attention to allocation
- Uses a linked list to keep track of free blocks
- Poor performance as the FS becomes fragmented
- No sorting of inodes or directories

Data Allocation in BSD FFS (1)

- BSD FFS uses knowledge of the physical structure of the disks
- Sequential blocks are allocated to rotationally sequential sectors:
  - next sector (allowing for interleave)
  - same cylinder
  - not necessarily the same track
- Performed well on devices at the time
- But: suffers from severe fragmentation overtime
- Modern drives (IDE, SCSI) are optimized for sequential access: FFS' performance is suboptimal for these drives

Data Allocation in BSD FFS (2)

- Divides disk into cylinder groups
- Intelligently allocates directories and inodes:
  - Directories are allocated in the group with the most free blocks out of all cylinder groups with the above-average free inodes
  - Inodes are allocated in the same cylinder group as their directory if possible
  - The first block of a file is allocated in its inode's cylinder group if possible
- Minimizes the effect of large files:
  - Forces block allocation to skip to next group every time a new indirect block is required
  - Thus, a single file cannot fill a complete cylinder group

Data Allocation in Viva FS

- Based on BSD FFS
- Attemps to allocate blocks sequentially instead of rotationally
- Maintains separate block bitmaps for:
  - single free blocks
  - clusters of 2, 4 and 8 consecutive free blocks
- Allocates a block's successor by looking for any free block within the next 32 blocks (uses a block from the bitmaps if that fails)
- Requests a single block for the first block, then 2, 4 and 8 blocks for subsequent requests

Data Allocation in Ext2fs (1)

- Combines:
  - FFS' principles of groups for free space
  - Viva's techniques for clustering blocks
- Block groups are used to cluster together related inodes and data (not necessarily related to the physical geometry of the disk)
- Uses Viva's "next-32-bits" search for allocating a block's successor
- If that fails, searchs forward:
  - within the group for an entire free byte in the bitmap (8 free blocks)
  - within the group for any free bit in the bitmap, if that fails
  - searchs subsequent groups in a similar manner, if that fails

Optimisations in File Systems - Stephen Tweedie 13

Data Allocation in Ext2fs (2)

- Preallocates up to 8 adjacent blocks when allocating a new block:
  - deallocates extra block on file close of if next allocation is not sequential
  - preallocation achieves good performances under heavy load
  - preallocation hit rates are around 75% even on very full file systems
- When a free byte bitmap (8 free blocks) is found, avoid creating unnecessary fragments by:
  - searching the previous byte
  - starting the allocation from the last free bit in that byte, if any

Optimisations in File Systems - Stephen Tweedie 14

Data Allocation in Ext2fs (3)

- Good locality:
  - of related files through block groups
  - of related blocks through 8-bits clustering of block allocations
- Avoids using up large blocks of free space of small files with a single bitmap
- Does not introduce many small fragments produced by Viva’s use of of small (2, 4 blocks) bitmaps

CPU Overhead Efficiency in Ext2fs

- Bitmaps are divided in block groups.
- Each block group has its own count of free blocks and inodes
- The size of bitmaps that must be searched is limited
- Block preallocation reduces allocation overhead by a factor of 4
- The physical and logical location of the last block is recorded in each inode
- Rapidly detects and deals with sequential allocations

Storage Efficiency in Ext2fs

- Fast symbolic links take up no data space:
  - the name of the target is stored in the inode itself, if it contains less than 64 characters
  - most symbolic links are short
  - 1 block is saved per fast symbolic link
- Optimised allocation routines allow small blocks to be used without suffering from severe performance penalties.

Reliability

- BSD FFS
- Viva FS
- Ext2fs

Reliability in BSD FFS

- Synchronous writes of metadata (inodes, directories, indirect blocks) ensure consistency of a directory tree after a crash
- Introduces `/lost+found` to recover unreferenced files on crash recovery
- Synchronous update is a major performance cost
- Synchronous writes only insure the integrity of metadata, but does not garantee consistency of files data
- Superblock data is replicated in each cylinder group

Reliability in Viva FS

- Keeps directory consistency by sorting writes to inodes/directories and bitmaps
- Maintains second bitmap in memory for inodes and blocks:
  - This bitmap is not commited to disk until all after inodes have all been commited
  - Blocks and inodes are not reused until their bitmaps have been commited
- Prevents directories and files sharing common inodes and blocks
- Fails to prevent inodes of indirect blocks referencing blocks not yet flushed to the disk (may still contain the contents of previously detected files)

Reliability in Ext2fs (1)

- Majority of crash recovery placed into the domain of e2fsck rather than into the kernel
- Directory tree is protected by giving metadata a higher priority for asynchronous writing (5 seconds timeout)
- Crashes are detected by a FS state:
  - a FS is marked as "dirty" once mounted read-write
  - a FS is marked as "clean" once fully unmounted
  - a FS is marked as "corrupt" when the Ext2fs kernel code detects inconsistencies, and the FS can be remounted in read-only mode
- Superblock and FS descriptors are replicated in each block group
- Bitmaps can occupy any contiguous blocks in a group

Reliability in Ext2fs (2)

- Powerful e2fsck and prioritized writes give good protection for data, in practice
- Schemes proposed to tighten reliability:
  - Ordered writes: metada are not commited until after the data they refernce have been written
  - Second bitmap to prevent reallocation of a data item until its erasure has been commited to disk
  - Consider when to write metadata for files which are continuously active, and which may never be completely flushed to disk

Fragmentation

- Is Fragmentation a Problem?
- Ext2fs and Fragmentation
- Performances Measurements
- The Defragmentation Process

Is Fragmentation a Problem?

- BSD FFS:
  - 90% is suggested as good maximum capacity
  - 95% full produces serious degradation
- Viva FS:
  - Experimentally, performance deterioration of no more than 50% is expected as the file system fills
- Extfs, SVR3:
  - Crippling fragmentation is a fact of life

Ext2fs and Fragmentation

- The defragmenter is complete and almost ready for release
- Experimentally, the need for a defragmenter is not obvious
- Effect of fragmentation on realistic performances may be complex:
  - sequential read access to very large files is not usually the critical performance problem
  - large files are often accessed randomly
  - very fast access to many small files is at least as commonly required as large sequential reads

Performances Measurements

Booting from the main root file system

	% full	Before Defrag	After Defrag	% cost of frag
Extfs	95%	37 sec	26 sec	42 %
Ext2fs	99%	40 sec	37 sec	8 %

Read throught the Linux source tree

	% full	Before Defrag	After Defrag	% cost of frag
Ext2fs	99%	53 sec	41 sec	29 %

Analysis of the Tests

- Boot test:
  - Despite the largely random mode of access requested, Extfs shows that fragmentation substantially degrades performances
  - Under Ext2fs:
    - » never defragmented in its 12-months life prior to this test
    - » a 210 MB filesystem suffers minimally from fragmentation
- Search test:
  - Even for major sequentially-bound disk I/O intensive tasks, this fragmented FS performs remarkably well in comparison to other file systems

The Defragmentation Process

- Highly optimised for speed of reorganisation
- Disk access are highly sequential
- Fragmentation is destructive: cannot be safely interrupted
- Structured in 4 distinct phases:
  - scan the inodes to collect informations
  - sort the inodes into order by their average data block, to reduce the average distance data blocks will travel during reorganisation
  - create the forward and reverse relocation by traversing the inodes again; updates inode metada
  - reshuffle the data area on disk from the relocation maps

The Reshuffle Phase (1)

- Maintains a buffer pool to store all reshuffled data
- Maintains two buffer types:
  - OUTPUT: block is read because it is about to be overwritten to its new location
  - RESCUE: block is read because it is about to be overwritten by a new block
- Sequentially processes the whole file system for writing:
  - for each disk block, find the appropriate existing block which is destined for this block
  - when the buffer pool is full, read the contents of each unread buffer, sort the buffers before the batch of reads, write out all OUTPUT buffers
  - buffers that have already been rescued may get written back soon, they migrate from RESCUE to OUTPUT. They are not written yet, it's cheaper to write them sequentially.

The Reshuffle Phase (2)

- If many data blocks have to move large distances, RESCUE buffers fill the pool:
  - when the pool reaches a limit, migrate all possible RESCUE buffers; if their destination is empty, make them OUTPUT buffers and write them as a batch
  - if that fails, select some RESCUE buffers to be "FORCE"d prematurely back to the disk in free blocks starting at the physical end of the disk
  - in the absence of FORCEing, all buffers get read and written exactly once as part of a sorted batch of I/O

Performances of the Defragmenter (1)

- Test made with:
  - the same 12-month old file system, 99% full
  - a 8MB buffer pool
- Defragmentation time: 35 minutes (15 minutes for 90 MB FS)
- Only 1566 blocks migrations, 8187 forces, out of 214938 writes
- Writes sorted in 72 batches, averaging 3MB per batch
- Subsequent defragmentations can be expected to batch more effectively (fewer data blocks will have to move long distances)

Results of the Defragmenter

- 25 out of 14120 inodes are fragmented
- Each fragmented file has on average 2 fragments
- Each fragment averages 319 blocks
- 12% of files left their inode’s native block group

Conclusion

- The Linux Ext2 file system contains many optimisations:
  - contiguous allocation, preallocation
  - space efficiency (fast symbolic links, small block size without performance penalty)
  - good error detection
- The Ext2fs tools are powerful:
  - excellent error recovery by an optimised checker
  - defragmenter