UNIVERSITY OF OSLO
Department of Informatics

Improving Disk I/O Performance on Linux

Master thesis

Carl Henrik Lunde

May 2009

Contents

1 Introduction **1**
1.1 Background and Motivation . . . 1
1.2 Problem Definition . . . 1
1.3 Main Contributions . . . 2
1.4 Outline . . . 3

2 Overview of Disk Storage **5**
2.1 The Rotating Disk Drive . . . 6
2.2 The I/O Scheduler . . . 9
2.2.1 Basic I/O Scheduling Algorithms . . . 9
2.2.2 Existing I/O Schedulers in Linux . . . 10
The Deadline I/O Scheduler . . . 10
Anticipatory I/O Scheduler (AS) . . . 12
Complete Fairness Queueing (CFQ) I/O Scheduler . 14
2.2.3 Performance of Linux Schedulers . . . 16
Analyzing and Visualization Tool . . . 16
Using the Visualization . . . 18
The Base Workload . . . 22
Deadline I/O Scheduler Performance . . . 23
Anticipatory I/O Scheduler (AS) Performance . . . 25
Complete Fairness Queueing (CFQ) Performance . . 27
2.3 The ext4 File System . . . 33
2.3.1 The Virtual File System . . . 33
2.3.2 Disk Layout . . . 34
Group Descriptors . . . 34
Inode and Block Bitmaps . . . 35
Virtual Block Groups (`flex_bg`) . . . 35
2.3.3 Inodes . . . 35
2.3.4 Directories . . . 36
2.3.5 Large, Indexed Directories . . . 36
2.3.6 Name Look-up . . . 39

2.3.7 Extents . . . 39
2.3.8 Allocation . . . 42
Inode Allocation . . . 42
Extent Allocation . . . 42
Dirent Allocation . . . 44
2.3.9 Journaling . . . 45
2.3.10 Readahead and Caching . . . 46
File Data Readahead . . . 46
Inode Readahead . . . 47
2.3.11 Implications of the File System Design . . . 48
Metadata / Inodes . . . 49
File Data Placement . . . 49
Evaluation of Data Ordering Cost . . . 49
2.4 Summary . . . 51

3 Adding Quality of Service (QoS) to CFQ 53
3.1 Design . . . 53
3.1.1 Non-intrusive . . . 53
3.1.2 Bandwidth (QoS) . . . 54
3.1.3 Request Deadlines (QoS) . . . 54
3.1.4 Work-conservation . . . 55
3.1.5 Global Throughput . . . 55
3.1.6 Admission Control . . . 55
3.2 Implementation . . . 56
3.2.1 Bandwidth Management . . . 56
3.2.2 Deadline Support . . . 58
3.2.3 Inter-process Elevator . . . 58
3.2.4 Queue Selection . . . 59
3.3 Evaluation . . . 60
3.3.1 Performance Limitations and Need for QoS . . . 60
3.3.2 Isolation . . . 64
Best Effort Readers Should Not Affect Reserved Readers . . . 64
Greedy Reserved Readers . . . 66
Other Potential Problems . . . 66
3.3.3 Work-conservation . . . 67
Over-provisioning . . . 67
Work-conservation with Non-Greedy Best Effort Readers . . . 68
3.3.4 Request Deadlines / Variable Round Support . . . 69
3.4 Conclusion and Future Work . . . 70

4 Scheduling in User Space **71**
4.1 Applicability of User Space Scheduling 71
4.2 Available Information 73
4.2.1 Metadata Placement 74
4.2.2 File Data Placement 75
4.3 Implementation of User Space Scheduling 77
4.4 Future Proofing Lower Layer Assumptions 77
4.4.1 Placement Assumptions 77
4.4.2 Rotational Disk 79
4.5 Case Study: The `tar` "Tape" Archive Program 79
4.5.1 GNU Tar 79
4.5.2 Design and Implementation 81
4.5.3 Testing and Evaluation 82
4.5.4 Verification by Visualizing Seeks 82
4.5.5 Partial Sorting 84
4.5.6 Aging vs. Improvement 85
4.5.7 File Systems, Directory Sizes and Aging 86
4.6 Discussion: Alternatives 89
4.7 Summary 91

5 Conclusion and Future Work **93**

A Creating a Controlled Test Environment **97**
A.1 The Aging Script 97
A.2 Aging Effects 98

B Source Code **101**

C List of acronyms **103**

Abstract

Motivated by increasing use of server consolidation, we have investigated the performance of disk I/O in Linux under various workloads. The tests identified several inefficiencies in the disk I/O scheduler, and that not all performance problems can be solved on the I/O scheduler layer. We have therefore attacked the problem on two levels: by modifying the I/O scheduler, and by developing user space techniques that reduce seeking for certain classes of applications. Our modifications to the I/O scheduler introduce a new priority class with QoS support for bandwidth and deadline requirements, while at the same time improving performance compared to the existing classes. The experimental results show that our I/O scheduler class can handle more multimedia streams without deadline misses, and that our user space techniques improve performance on workloads such as file archiving. With our experiments we have found that the number of concurrent media streams can be increased by 17%, and that archiving tasks ran almost 5 times faster in the best case. Our limited set of experiments has not revealed any performance disadvantages of the proposed techniques.

Acknowledgments

I would like to thank everyone at the Simula lab, especially Håvard Espeland and Željko Vrba, PING and my supervisors Pål Halvorsen and Carsten Griwodz for the help, ideas and interesting conversations. I would also like to thank my employer and colleagues for allowing me to have a flexible work schedule.

Oslo, May 2009
Carl Henrik Lunde

Chapter 1

Introduction

1.1 Background and Motivation

Improvements in latency and bandwidth of disks lag that of CPU, RAM and Internet connections, while at the same time the disk capacity increases at a higher rate than the bandwidth and latency improves [1]. These facts put more pressure on the disk than ever before, and it also opens the opportunity to spend more of resources such as RAM and CPU on reaching the latency and throughput requirements from the disk.

Today, the Linux kernel is used for a wide variety of tasks, and recently is has become more popular to consolidate multiple applications into fewer servers. Consolidation may yield cost and environmental savings [2], but it also introduces new challenges as shared resources become more contended than before, and the amount of data on a single disk increases. Because the applications serve different purposes, they have different requirements. For example: a video has strict requirements for latency and bandwidth, an image for a web page must be returned within a short amount of time, while batch jobs such as backups and other tasks do not have any special requirements, but they should share resources fairly.

1.2 Problem Definition

In this thesis, we will study how we can improve the performance of I/O on rotating disk devices, while maintaining latency requirements of multimedia. The current Real-Time (RT) scheduling class in Linux does not have any knowledge of latency requirements, and must therefore dispatch requests in First-In, First-Out (FIFO) order. By allowing applications to specify latency requirements we can delay requests if it yields a better I/O

schedule. An additional problem with the Real-Time (RT) class is that any misbehaving, or simply work conserving task, may starve the system. In a consolidated environment it is even more important to avoid this situation, as the number of affected applications increases.

We cannot require all users and applications to specify bandwidth and latency requirements, which means that it is also important to support a fair best effort class. Having a special purpose QoS scheduler is not acceptable.

Existing implementations of I/O schedulers tend to focus on one type of load, and not respect the requirements of general purpose systems, such as fairness. This is less user friendly and leaves the user with an additional task, choosing an I/O scheduler. By adapting an existing scheduler we aim to avoid this problem.

While much work already has been done on the I/O scheduler level, we will also take a look at all the other layers involved in disk I/O, to see if there are possibilities for multi-layer improvements. Specifically, we want to study the I/O patterns for applications which reads multiple files, but never have more than one I/O request pending. It is not possible for the I/O scheduler to improve the performance of such applications. Recent changes in file system programming interfaces expose more information about data location, which provide interesting opportunities. Typical applications for this includes backup software and data indexing software.

1.3 Main Contributions

We show that there is still potential for improving the performance of I/O bound tasks in GNU/Linux, on both the I/O scheduler layer and the application layer. We have made a tool for visualizing I/O from multiple perspectives at the same time, which we consider valuable for I/O scheduler development and analysis. We show that a general purpose I/O scheduler can be modified to better support hard latency and bandwidth requirements from multimedia, while at the same time improving performance and maintaining the original properties.

Finally, we have shown that taking I/O scheduling to another layer is possible, and that some specialized tasks such as backup software can improve throughput by using this technique.

1.4 Outline

In chapter 2 we give a detailed analysis of the I/O schedulers shipped with Linux 2.6.29, how the file system works, and how the file system and user space applications limit the options of the I/O scheduler. We provide a new visualization of the interactions between the layers involved in I/O. The goal is to find potential improvements and issues which we can study further.

In chapter 3 we design, implement and evaluate what we consider a practical solution which addresses some of the concerns we have with the existing I/O scheduler solution available in Linux as written in chapter 2.

In chapter 4 we look at how applications may mitigate the problem found in chapter 2, and we design and implement a solution. Finally, in chapter 5, we provide a short conclusion and summary of the thesis.

Chapter 2

Overview of Disk Storage

In this chapter, we will take an in-depth look at the layers involved in disk input/output in a typical GNU/Linux installation. Our goal is to identify problems related to system performance and QoS for multimedia streams. We believe it is not possible to look at only one layer in order to understand and improve the current situation, so we will not only study the disk and I/O scheduler, but also the file system.

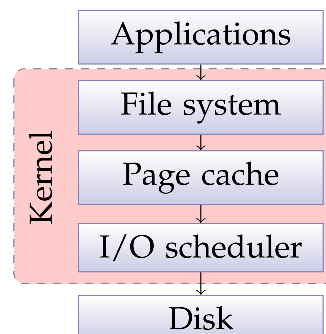


Figure 2.1: Layers involved in block I/O

As shown in figure 2.1, there are five components that heavily influence the I/O schedule:

- The *applications* running in user space. They initiate I/O transfers through system calls such as `read()` and `write()`, and by accessing memory pages mapped to files.
- The *file system* which translates the requests from user space into one or more block I/O requests. The file system also implements block allocation strategies, which determine the physical placement on disk and potential fragmentation.

- The *page cache* which is responsible for caching the set of disk blocks which are most likely to be used again.
- The *I/O scheduler* which sorts, merges and prioritizes the I/O requests and dispatches them to the disk.
- The *disk* which services the requests. It often provides another layer of caching. With advanced features such as Native Command Queuing (NCQ)/Tagged Command Queuing (TCQ) disks may queue several requests, and reorder them to minimize seeks and increase performance. The behaviour is vendor-specific.

To limit the scope of this thesis we will not study the subject of caching.

2.1 The Rotating Disk Drive

Since the personal computer was introduced, and still today, the most popular storage medium has been rotating hard disk drives, with data stored on both surfaces of the platters on a rotating spindle.

The disk drive is divided into multiple areas; heads, cylinders and sectors, as shown in figure 2.2. A cylinder consists of the tracks with the same diameter on all the platters. A set of arms are locked together, with one arm for each surface. An arm reads and records data to any position on a surface by moving the heads to the correct cylinder, and then wait for the sector in question to arrive under the head due to disk rotation. Data is written as 0 or 1 bits by magnetizing the platter, and read back by sensing magnetism.

Because the track density decreases as the diameter increases, drives are often divided into zones with an increasing number of sectors per track. As mentioned, three steps must be performed for a sector to be read or written, and the execution time is the sum of the three steps:

Seek time The time required for the arm to settle at the right track. This is determined by seek distance, disk diameter and disk arm performance.

Rotational delay The time required for the right sector to appear under the head. Rotational delay is determined by spindle speed.

Transfer time How long it takes for the sector to be read. This is determined by the density at the given zone, and spindle speed.

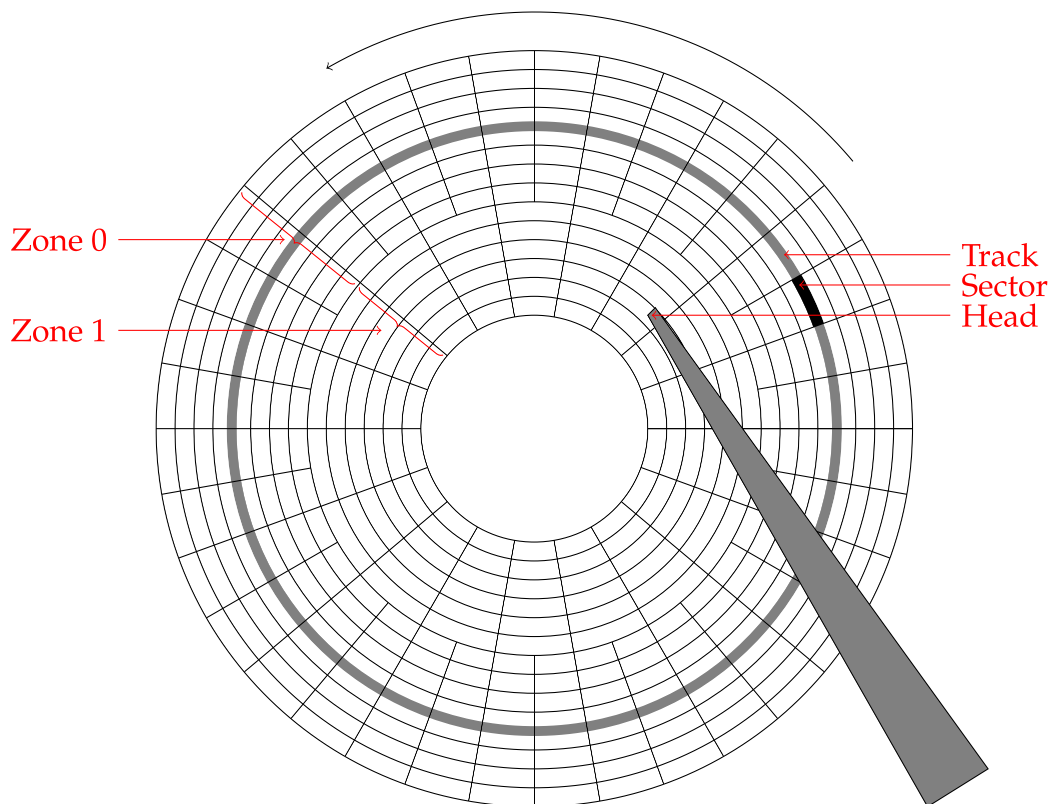


Figure 2.2: Rotating disk drive

The dominating factors are the seek time and rotational delay. For a random read operation, we must assume half a rotation in rotational delay, and the average value of all possible combinations of seek lengths. A modern high performance 15000 RPM server disk drive like the Seagate Cheetah 15k.6 [3] has a specified average read time of 3.4 ms. For this drive, we can calculate the average rotational delay to 2 ms as shown below:

$$\frac{60 \text{ s/min} \cdot 1000 \text{ ms/s}}{15000 \text{ RPM}} \cdot \frac{1}{2} = 2 \text{ ms}$$

Desktop drives (typically 7200 RPM) and laptop drives (often 5400 RPM) are slower, with average read times ranging from 14 ms to 8.5 ms. The transfer time for a contiguous read, however, is another matter. When the arm is aligned correctly, each track may be read at the speed at which the platters rotate. Between each track there is a slight delay for the arm

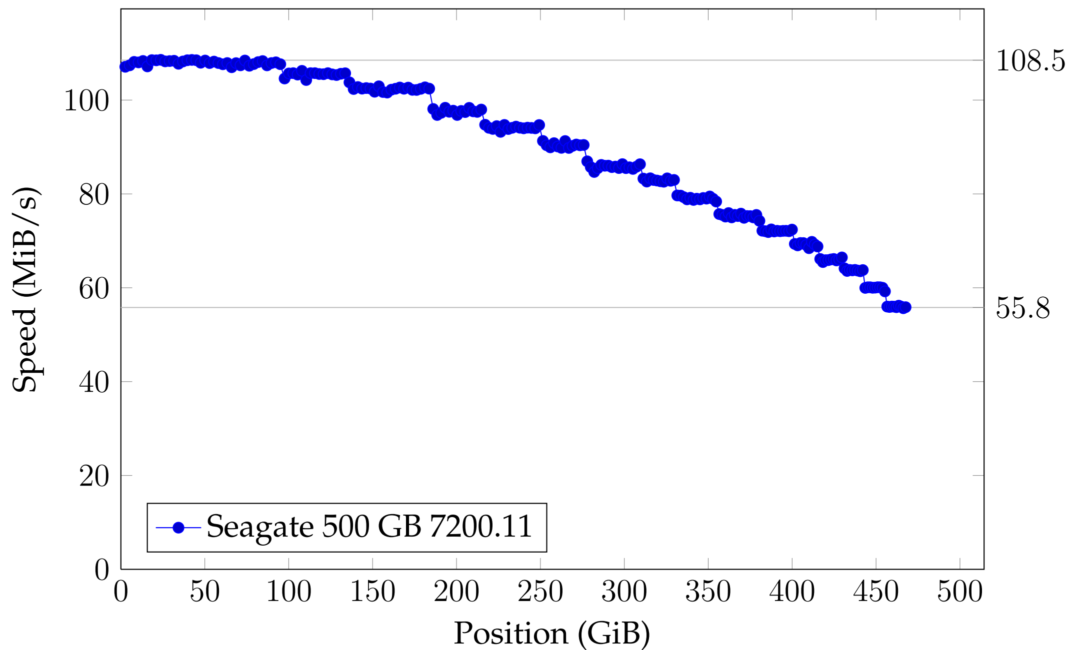


Figure 2.3: Performance effects of zoning on 3.5" rotating disk

to re-align. As a consequence of the zones, the maximum transfer rate is often higher on the outer tracks of the disk, because the platter rotates at a constant speed. The logical address of the outer track is lowest, which means that the beginning of the disk is stored on the fastest (outer) track. We have measured the effects of zoning on the disk in our test environment, the results can be seen in figure 2.3. Based on this figure we believe our disk has 16-17 zones. Vendors typically specify sustained data rates of 50-120 MiB/s, depending on form factor, rotational speed and data density.

In our test environment, we have a Seagate Barracuda 7200.11 disk [4] (model number ST3500320AS). A summary of the specifications from the product manual is shown in table 2.1. This is a workstation-class disk, connected to the computer through a SATA bus at 3 Gb/s speed. The computer is a Mac Pro running Linux, and it has 10 GB RAM, and two Intel Xeon Quad Core CPUs running at 2.8 GHz. No tests in this thesis will rely on caching or CPU power.

Drive specification	
Sustained data transfer rate OD	105 Mbytes/sec max
Average latency	4.15 ms
Track-to-track seek time (read)	$<$ 0.8 ms typical
Average seek (read)	8.5 ms typical
Heads	4
Disks	2
Spindle speed	7200 RPM
Cache buffer	32 MiB
Capacity	465 GiB

Table 2.1: Drive specification

2.2 The I/O Scheduler

The next component in the I/O path (figure 2.1) is the I/O scheduler, and all I/O requests to the disk pass through it. The scheduler implements policies which determine when I/O requests should be dispatched to the disk, and in which order. The I/O scheduler must make a balanced choice between global throughput, latency and priority.

As shown in the previous section, servicing one individual random request may take as long 10 milliseconds, while during 10 milliseconds a modern computer Central Processing Unit (CPU) runs 30 million clock cycles. This means that there could be an opportunity to spend some CPU time to plan the I/O to improve overall system performance, and that is one of the tasks of the I/O scheduler. The other task is to implement a policy for how the shared disk resource should be divided amongst the processes on the system.

2.2.1 Basic I/O Scheduling Algorithms

The simplest I/O scheduler simply forwards each request to the disk in the same order as they arrive, i.e., First-Come, First-Served (FCFS) [5]. The performance with FCFS suffers because it does not attempt to reduce seeks. In the Linux kernel, a variant of this strategy is implemented as the No Operation (NOOP) I/O scheduler. It is not truly an FCFS scheduler as requests may be merged with other requests ahead in the queue, if they are physically consecutive. The NOOP I/O scheduler is not designed for rotating hard disks, but for random access devices such as Solid State Drives (SSDs).

Next in the evolution chain is Shortest Seek First (SSF) [5], which, as

the name implies, always picks the request which is closest to the current request. This reduces average response time considerably. The problem with SSF is that it is tends to be very unfair: If requests are random all over the disk, and the system is heavily loaded, we may easily end up with a situation where the requests near both edges of the disk may never be served, or at least get long service times.

A fairer alternative to SSF is the elevator algorithm [5]. The first variant is SCAN [5], which services requests in one direction at a time. First, requests are serviced only if they have a sector number larger then the previously serviced requests. When there are no more requests to serve with a larger sector number, the direction is reversed. SCAN does also have a tendency to serve requests near the middle of the disk faster and more often, so a fairer alternative may be needed. Circular SCAN (C-SCAN) only services requests in one direction, which sacrifices some performance for improved fairness. When the queue depth is very large, C-SCAN performs better than SCAN.

2.2.2 Existing I/O Schedulers in Linux

The Linux 2.6 kernel has a pluggable scheduler framework. Each block device has its own scheduler, and it can be changed at runtime by writing the scheduler name to a special *sysfs* file, called `/sys/block/<blockdev>/queue/scheduler`.

This allows the user to select different schedulers depending on the properties of the block device, e.g., a user may want the NOOP scheduler for a flash disk and the anticipatory scheduler for a CD-ROM player. The schedulers have several tunable parameters, and in this section, we will highlight them by writing in a fixed width font such as "`back_seek_max`", which means that there is a tunable parameter for the scheduler in `/sys/block/<blockdev>/queue/iosched/back_seek_max`. As with choosing a scheduler, the parameters may be tuned during runtime by writing to the special file.

The Deadline I/O Scheduler

The deadline I/O scheduler [6] is a C-SCAN based I/O scheduler with the addition of soft deadlines to prevent starvation and to prevent requests from being delayed too long. The scheduler maintains two queue types:

The elevator is a queue sorted by sector, and serviced mostly like a C-SCAN elevator. It is implemented by a Red-Black tree [7, 8], a self-

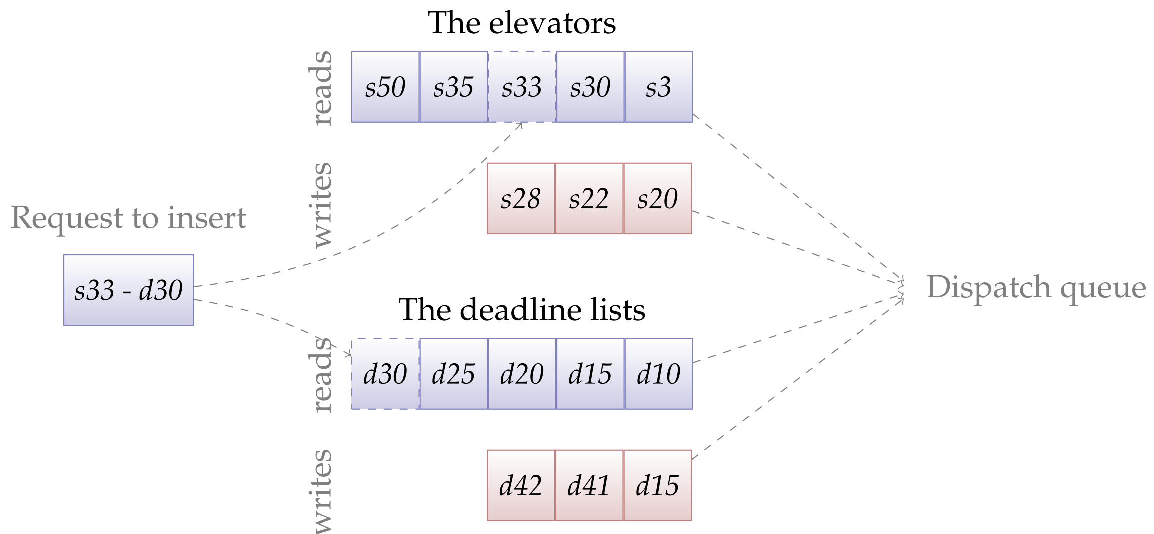


Figure 2.4: Queues in the Deadline I/O Scheduler

balancing binary search tree.

The deadline list, a FIFO queue for deadlines implemented as a standard Linux kernel linked list [9]. The deadlines are only implemented to prevent starvation, and do not allow service differentiation, because all processes get the same deadline. This allows for fast $O(1)$ insertion in the deadline FIFO queue, because requests are always appended to the tail of the list.

Each I/O request is put in both the elevator queue and the deadline FIFO, and there are two instances of each queue, one for each data direction (read/write), which allows the scheduler to prioritize reads and writes differently. In figure 2.4, we have shown how a read request at sector 33 with deadline 30 would be appended to the read FIFO queue, and inserted somewhere in the middle of the read elevator, sorted by sector.

Normally, one of the elevator queues is serviced, up to 16 requests in a row (tunable with `fifo_batch`). When the batch is over, or there are no more requests to batch from the current elevator, a new data direction is chosen. The data direction defaults to reads—writes are only serviced if there have been pending writes two times (`writes_starved`) when choosing a data direction.

After choosing data direction, the head of the FIFO deadline queue is checked for deadline expiry. If there are no expired deadlines in the

chosen direction, the previous batch will be continued if possible, i.e., if the current data direction is the same as for the previous batch. Otherwise, the deadline FIFO will be conferred to, and a new batch will be started (from the sorted list) at the request with the shortest deadline.

By default, the deadline for a write request is 5 seconds (`write_expire`), while a read request has a deadline of 0.5 seconds (`read_expire`), and the read queue is serviced up to two times before the write queue is considered. The reason for this clear preference of reads is that writes easily can starve reads [6], and writes are usually asynchronous, while reads are synchronous and blocking the process waiting for the result.

In addition to ordering requests, another important task of Linux I/O schedulers is request merging. The file systems and Virtual File System (VFS) layer in Linux tend to split large `read()` operations from user space into smaller requests, often one file system block at a time, when sending them to the I/O scheduler. The I/O scheduler may merge these smaller requests into larger requests before dispatching them to the disk. A request may be merged with a request from a different process, but only if the data direction (read/write) is the same, and there may not be a gap between the two original requests. The Deadline I/O Scheduler uses the sorted queues to check if merging is possible.

The Deadline I/O Scheduler does not have any QoS support: I/O deadlines are global, so it is not possible to prioritize one process over another. Although it is possible to change the priority of reads relative to writes, this is usually not sufficient. It is also important to note that the deadlines are soft, and exist only to prevent starvation.

Anticipatory I/O Scheduler (AS)

The Anticipatory I/O Scheduler [6, 11] tries to solve a problem with the Deadline I/O Scheduler; the global throughput is often low, especially in streaming scenarios, and when there is high spatial locality. The anticipatory scheduling algorithm was introduced in [10], and it blames the low global throughput on trashing, due to so called “deceptive idleness”. The problem with the Deadline I/O Scheduler is that it is forced to make decisions too early. To solve this, the Anticipatory I/O Scheduler introduces small delays before dispatching seeking requests. This is to make sure the current process gets time to dispatch a new request after processing the latest completed request. Then the scheduler gets full view of all possible options, which often results in a better schedule.

When a request is completed, the scheduler checks the sorted request queue. If the next request in the elevator is either from the same process as

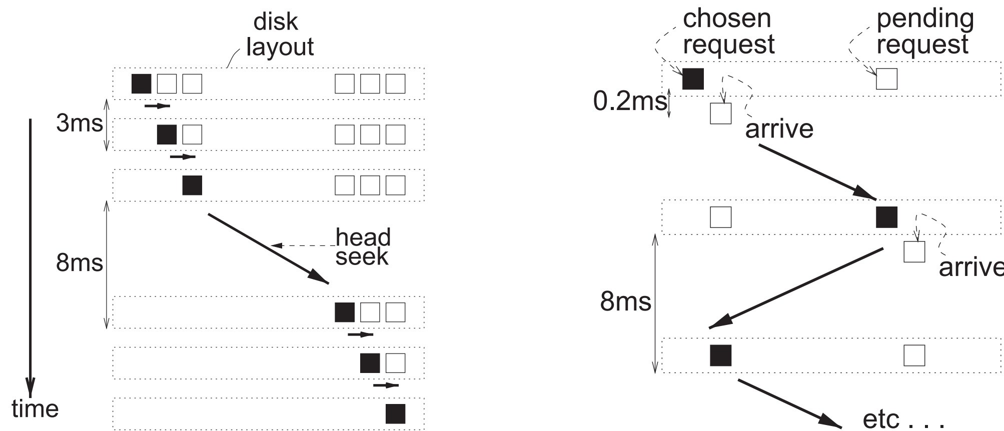


Figure 2.5: (a) anticipatory scheduling (b) trashing due to deceptive idleness [10]

previously serviced, or the next request is within a few thousand sectors, it will be serviced directly. Otherwise, the scheduler confers to the process' I/O interval statistics: if the process is likely to issue a new I/O request shortly, the scheduler waits a small amount of time (`antic_expire`, default 6 ms), in anticipation of a new I/O request close to the current disk head position. This solves the trashing problem with the Deadline I/O Scheduler as shown in figure 2.5.

The queues in the Anticipatory I/O Scheduler are the same as shown in figure 2.4. In addition to the anticipation, the Anticipatory I/O Scheduler is different from the Deadline I/O Scheduler because the elevator is not as strict: backwards seeks are allowed, but only if they are relatively short.

There may be many reasons for deceptive idleness, where waiting for the next request is likely to be beneficial, such as:

VFS think time The function `mpage_readpages` is used by many Linux file systems for multi-block buffered I/O, such as through the `read` system call. The function ends up calling `submit_bio` many times in a loop, which means the I/O scheduler only sees a part of the picture when the first I/O request arrives, but shortly after one or more consecutive requests will arrive.

File system think time The file system may need to look up indirect blocks or extent maps to get the physical location of the next logical block in a large `read` request. This means that the process is blocked doing synchronous I/O, and the file system needs a few microseconds to parse the data before the next request can be issued.

Process buffering and data dependencies Many processes issue small reads at a time, parse the result, and continue reading. This may be because the data needs to be parsed to know when to stop or what data to read next, or because the process wants to keep a low memory profile. Another example is processes reading multiple files in the same directory, if the file system allocator has done a good job, they should be located close to each other.

CFQ I/O Scheduler

Figure 2.6: Priority classes in CFQ

The CFQ I/O scheduler is the default I/O scheduler in Linux, and it tries to satisfy several goals:

- Keep complete fairness among I/O requests in the same class by assigning time slices to each process. Fairness is on a time basis, not throughput, which means that a process performing random I/O gets lower throughput than a process with sequential I/O.
- Provide some level of QoS by dividing processes into I/O contexts in different classes, as shown in figure 2.6.
- Give relatively high throughput, the assumption is that requests from an individual process tend to be close together (spatial locality). By waiting for more requests from the same process the scheduler hopes to avoid long seeks.
- Latency is kept proportional to system load by scheduling each process periodically.

The CFQ I/O Scheduler maintains queues *per process*, and each process is served periodically. The active process gets exclusive access to the underlying block device for the duration of the time slice, unless preempted. This is an alternative approach to solving the problem with deceptive idleness. The length of the time slice is calculated by the *priority level*, while

Priority	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Slice length	40	60	80	100	120	140	160	180
Slice offset, busy=4	420	360	300	240	180	120	60	0
Slice offset, busy=5	560	480	400	320	240	160	80	0

Table 2.2: Varying number of busy queues and priority level decide slice length and offset. Values are calculated for the default `base_slice`, 100 ms

the period also depends on the number of other processes with pending requests, as shown in table 2.2.

A process may belong to one out of three I/O priority classes:

Real-Time (RT) Processes from the RT scheduling class are giving first access to the disk every time. The RT class has eight priority levels, which control how the disk time is divided between the RT class processes, with 0 being the highest and 7 the lowest priority. Because RT processes may starve processes with lower priorities, only the root user may set this priority level.

Best Effort (BE) The BE I/O class is serviced when there are no RT requests queued, and as with RT there are eight priority levels (0-7). Priority four is default, but if a specific CPU priority is set, the CPU priority may map to a lower or higher I/O priority. The BE scheduling class is the default for new processes. If the RT class always has pending requests, the BE class will be starved.

Idle When the RT and BE queues are empty, requests are picked from the idle class. Because the idle class may be starved on heavily loaded systems, only the root user may assign processes to the idle class.

The CFQ Service Tree is a list of all the active queues (processes). It is first ordered by I/O class (RT > BE > Idle), and then expected service time (when the time slice should start, approximately). The service time is merely a guideline, not a hard deadline. As can be seen in table 2.2, if there are four busy queues with the default priority and default time slice, they will each be assigned a 100 ms time slice, and a 240 ms deadline—but if all processes are busy the scheduling time will be 300 ms, so some overbooking is done. A process does not preempt another process when its deadline expires.

The scheduler is work conserving; if the scheduler considers the active process as idle, it will prematurely end the time slice, store the residual time, and move on to the next process. A time slice in the BE class may also

be preempted if a request arrives in the RT class. For both cases of preemption, the next time the original process is scheduled it will be scheduled a little earlier the next round[1].

For each process, requests are stored much like in the Deadline I/O Scheduler (figure 2.4), but instead of separating writes from reads, CFQ separates synchronous requests from asynchronous request[2]. The sorted tree is serviced as long as the deadline of the request at the head of FIFO queue has not expired.

It should be noted that older literature [6] describes a different implementation of the CFQ I/O scheduler, which used a staircase round robin service of 64 queues. A process was assigned to a queue by hashing the thread group identifier. The algorithm we describe was introduced in Linux 2.6.22 and is up to date as of 2.6.29.

2.2.3 Performance of Linux Schedulers

Analyzing and Visualization Tool

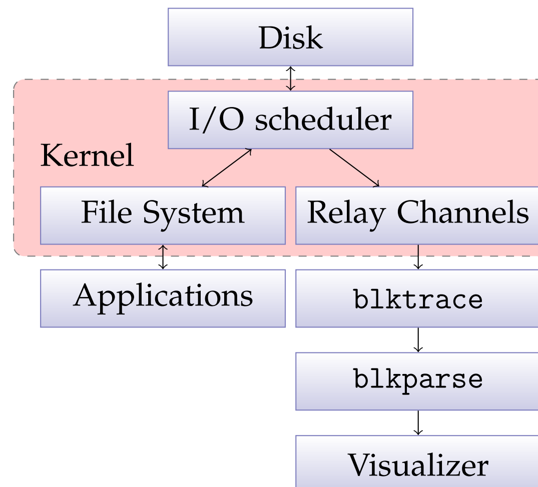


Figure 2.7: blktrace and visualization tool

As we delve into the analyzing of the I/O schedulers available in the Linux-kernel, we will show some visualizations of how they service requests, with a focus on how the performance is seen from user space ap-

[1]This functionality is currently not working

[2]Typically, writes are asynchronous, and the CFQ I/O scheduler considers all reads as synchronous

plication. We have developed a program for visualizing the events happening in user space, in the kernel, and in the disk.

Our goal is to quickly observe the events in each layer, how they interact and also to provide enough information to study even small details such as NCQ. To gather information from these components, we have used the existing `blktrace` mechanism [12, 13], with some modifications. `blktrace` is a block layer I/O tracing tool written by Jens Axboe, the current maintainer of the Linux Block Layer.

`blktrace` has defined many trace points within the kernel block layer code and inside the I/O schedulers. They remain inactive until activated by an `ioctl` system call by a user space client. When active, the trace points emit events to user space through a low overhead mechanism called the relay file system. Data is transferred to user space as binary data streams, one per CPU, and is later interpreted by the `blkparse` utility.

There are roughly 20 different events emitted by `blktrace`, of which we only use a few. Events are identified by a single character, such as `D` for "dispatch". Below we list the events we use, for a full list refer to the `blkparse(1)` manual page [14].

- Request queued (`Q` in `blkparse`). We use this to track I/O request ownership, because other events, such as dispatch and completion, run in an arbitrary process context.

- I/O scheduler request dispatch (`D` in `blkparse`). This event is issued when the I/O scheduler moves a request from a queue inside the I/O scheduler and to the dispatch queue, so that the disk can service the request.

- Disk I/O completion (`C` in `blkparse`). This happens when the disk has serviced a request. For a write operation this means that the data has been written to the platter (or at least buffered), and for a read operation that the data is stored in host memory.

- Virtual File System (VFS) operations such as `read()` and `write()`. We have patched `blktrace` to emit these events as text messages (`m` in `blkparse`), because we consider them very relevant to understand the I/O schedule from an application point of view. `blktrace` originally only supports events happening on request queues.

As we can see in listing 2.1, the `blkparse` output is very verbose. The first 10 lines show that process 8593 enters the `read()` system call, and queues 512 blocks[3] of data to read, starting at sector 831107423. At 2.01

[3] In this context, a block is 512 bytes

the disk returns 256 blocks from sector 33655111, and the application 8598 queues another 256 block request at 33655623, which is dispatched immediately. This cycle repeats until time 2.0158, where we see the `read()` system call has completed for process 8598. The process almost instantly enters another `read()`, which is sequential to the previous request, so the I/O scheduler dispatches the request immediately. The first process, 8593, is still blocked and the trace does not show how long it had to wait.

The point of this example is to show that `blkparse` is very useful to study the finer details of the I/O scheduler, but it is not suited to get an understanding of the big picture, where patterns may take seconds to emerge.

Using the Visualization

In figure 2.8, we present an annotated graph which is created by our visualization tool. We get a view of four seconds showing how the I/O scheduler dispatches I/O requests for each process, how the disk head must move and for how long each process is blocked. This is the same load we will use for all schedulers. While the scenario may be a bit special, it is well suited to highlight the behaviour of each scheduler. Numbers in parentheses below reference the same number in the figure 2.8, e.g., (1).

Process information (0) We provide information about the process name, average bandwidth, requested bandwidth and request size. The process name indicates the type of job, S_n indicates a stream with contiguous requests, while R_n is a process doing random reads. MF_n is a multi-file reading process, which does file system calls such as `open()`, `readdir()`, `read()` and `stat()`. As the multi-file process is always best effort, we provide information about average bandwidth and the number of files processed.

`read()` histogram (1) There are two histograms for each process, located to the right of the process information. The first histogram we have highlighted is the `read()` histogram, which shows how long the VFS and I/O scheduler uses to serve the `read()` requests. During this time the process is blocked, and from the application point of view this is the only thing that matters, not how each individual sector is handled.

Dispatch-to-Completion (D2C) histogram (2) The next histogram shows the individual block request completion times, the time from a request is

```
Blockdev CPU Sequence    Time         PID   Event Sector + blocks
  8,48   3         0     2.004027488  8593  m     N enter vfs_read
  8,48   3       657     2.004047581  8593  A     R 831107423 + 256 <- (8,49) 831107360
  8,48   3       658     2.004047704  8593  Q     R 831107423 + 256 [iosched-bench]
  8,48   3       659     2.004048864  8593  G     R 831107423 + 256 [iosched-bench]
  8,48   3       660     2.004049761  8593  I     R 831107423 + 256 [iosched-bench]
  8,48   3       661     2.004155593  8593  A     R 831107679 + 256 <- (8,49) 831107616
  8,48   3       662     2.004155696  8593  Q     R 831107679 + 256 [iosched-bench]
  8,48   3       663     2.004156385  8593  M     R 831107679 + 256 [iosched-bench]
  8,48   3       664     2.004157215  8593  U     N [iosched-bench] 7
  8,48   2       599     2.010858781  8598  C     R 33655111 + 256 [0]
  8,48   2       600     2.010983458  8598  A     R 33655623 + 256 <- (8,49) 33655560
  8,48   2       601     2.010983593  8598  Q     R 33655623 + 256 [iosched-bench]
  8,48   2       602     2.010984420  8598  G     R 33655623 + 256 [iosched-bench]
  8,48   2       603     2.010985094  8598  I     R 33655623 + 256 [iosched-bench]
  8,48   2       604     2.010988047  8598  D     R 33655623 + 256 [iosched-bench]
  8,48   2       605     2.010990666  8598  U     N [iosched-bench] 7
  8,48   1       530     2.012061299  8598  C     R 33655367 + 256 [0]
  8,48   1       531     2.012182151  8598  A     R 33655879 + 256 <- (8,49) 33655816
  8,48   1       532     2.012182705  8598  Q     R 33655879 + 256 [iosched-bench]
  8,48   1       533     2.012183893  8598  G     R 33655879 + 256 [iosched-bench]
  8,48   1       534     2.012184605  8598  I     R 33655879 + 256 [iosched-bench]
  8,48   1       535     2.012187585  8598  D     R 33655879 + 256 [iosched-bench]
  8,48   1       536     2.012190294  8598  U     N [iosched-bench] 7
  8,48   3       665     2.012834865  8598  C     R 33655623 + 256 [0]
  8,48   3       666     2.012948913  8598  A     R 33656135 + 256 <- (8,49) 33656072
  8,48   3       667     2.012949078  8598  Q     R 33656135 + 256 [iosched-bench]
  8,48   3       668     2.012949913  8598  G     R 33656135 + 256 [iosched-bench]
  8,48   3       669     2.012950555  8598  I     R 33656135 + 256 [iosched-bench]
  8,48   3       670     2.012953502  8598  D     R 33656135 + 256 [iosched-bench]
  8,48   3       671     2.012956161  8598  U     N [iosched-bench] 7
  8,48   4       579     2.013823914  8598  C     R 33655879 + 256 [0]
  8,48   4       580     2.013944217  8598  A     R 33656391 + 256 <- (8,49) 33656328
  8,48   4       581     2.013944375  8598  Q     R 33656391 + 256 [iosched-bench]
  8,48   4       582     2.013945518  8598  G     R 33656391 + 256 [iosched-bench]
  8,48   4       583     2.013946170  8598  I     R 33656391 + 256 [iosched-bench]
  8,48   4       584     2.013949706  8598  D     R 33656391 + 256 [iosched-bench]
  8,48   4       585     2.013952260  8598  U     N [iosched-bench] 7
  8,48   5       569     2.014811161  8598  C     R 33656135 + 256 [0]
  8,48   5       570     2.014930361  8598  A     R 33656647 + 256 <- (8,49) 33656584
  8,48   5       571     2.014930489  8598  Q     R 33656647 + 256 [iosched-bench]
  8,48   5       572     2.014931361  8598  G     R 33656647 + 256 [iosched-bench]
  8,48   5       573     2.014932053  8598  I     R 33656647 + 256 [iosched-bench]
  8,48   5       574     2.014935063  8598  D     R 33656647 + 256 [iosched-bench]
  8,48   5       575     2.014938063  8598  U     N [iosched-bench] 7
  8,48   6       652     2.015804419  8598  C     R 33656391 + 256 [0]
  8,48   6       653     2.015926600  8598  A     R 33656903 + 256 <- (8,49) 33656840
  8,48   6       654     2.015926798  8598  Q     R 33656903 + 256 [iosched-bench]
  8,48   6       655     2.015927800  8598  G     R 33656903 + 256 [iosched-bench]
  8,48   6       656     2.015928484  8598  I     R 33656903 + 256 [iosched-bench]
  8,48   6       657     2.015931482  8598  D     R 33656903 + 256 [iosched-bench]
  8,48   6       658     2.015934236  8598  U     N [iosched-bench] 7
  8,48   0       624     2.016789937  8598  C     R 33656647 + 256 [0]
  8,48   0         0     2.016891686  8598  m     N leave vfs_read
  8,48   0         0     2.016908007  8598  m     N enter vfs_read
  8,48   0       625     2.016928340  8598  A     R 33657159 + 256 <- (8,49) 33657096
  8,48   0       626     2.016928466  8598  Q     R 33657159 + 256 [iosched-bench]
  8,48   0       627     2.016929298  8598  G     R 33657159 + 256 [iosched-bench]
  8,48   0       628     2.016930012  8598  I     R 33657159 + 256 [iosched-bench]
  8,48   0       629     2.016933714  8598  D     R 33657159 + 256 [iosched-bench]
  8,48   0       630     2.016936375  8598  U     N [iosched-bench] 7
  8,48   2       606     2.019041130  8598  C     R 33656903 + 256 [0]
  8,48   2       607     2.019164273  8598  A     R 33657415 + 256 <- (8,49) 33657352
  8,48   2       608     2.019164441  8598  Q     R 33657415 + 256 [iosched-bench]
  8,48   2       609     2.019165180  8598  G     R 33657415 + 256 [iosched-bench]
  8,48   2       610     2.019165850  8598  I     R 33657415 + 256 [iosched-bench]
  8,48   2       611     2.019168867  8598  D     R 33657415 + 256 [iosched-bench]
  8,48   2       612     2.019171584  8598  U     N [iosched-bench] 7
  8,48   1       537     2.020029800  8598  C     R 33657159 + 256 [0]
```

Listing 2.1: 20 ms of `blkparse` output

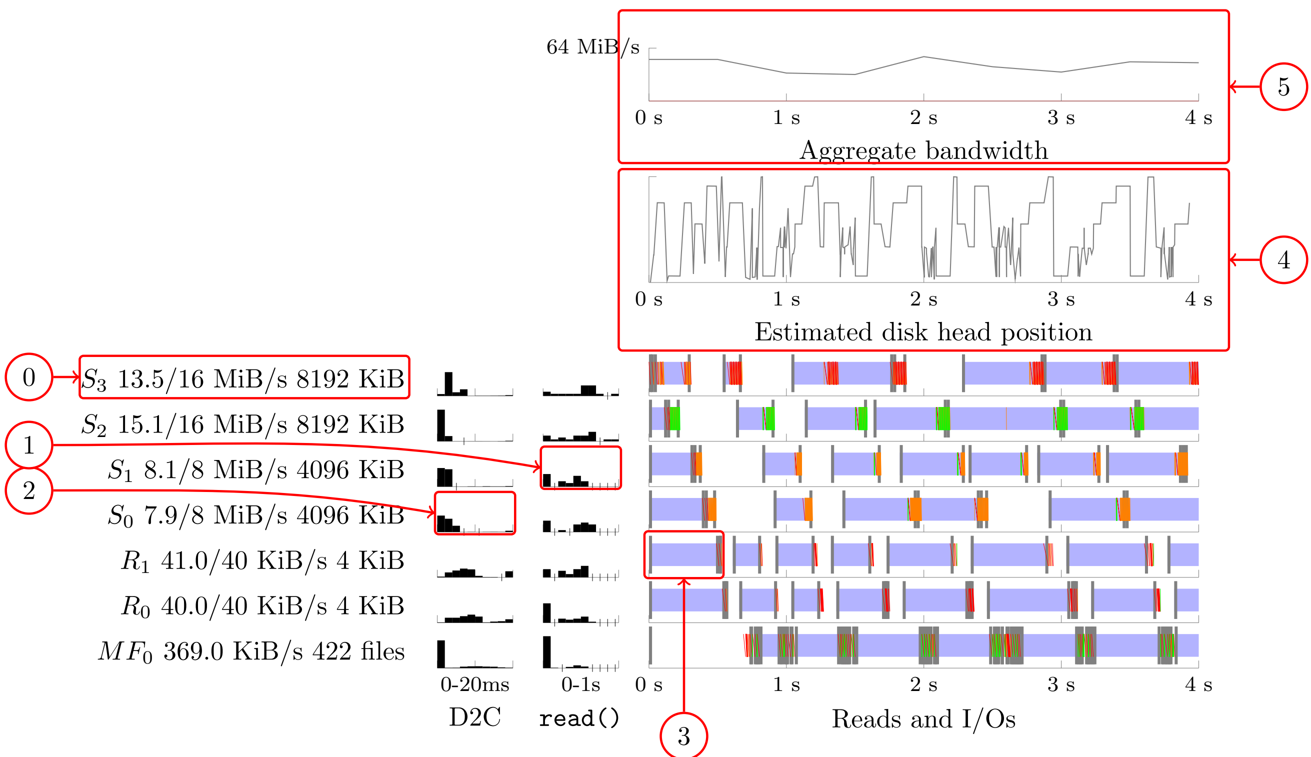


Figure 2.8: Annotated visualization

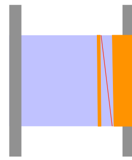


Figure 2.9: Read system call and I/O requests

dispatched (sent to the disk) until it has completed (data is ready). It does not include the time spent in queues inside the I/O scheduler.

A left-weighted histogram indicates that mosts requests are sequential and require few seeks. The histogram is limited to 20ms, any request with a longer D2C time is shown as 20 ms. This is mostly seen when NCQ is enabled. Note that request sizes are not taken into consideration for these histograms.

VFS I/O system calls (3) On the right hand side there is a time line for each process. Upon entering a system call, a gray vertical line is drawn, and while the process is blocked we render the body blue. When the requests are done, another gray vertical gray line is drawn. This is the user space perspective on the schedule. Typical blocking calls are `read()`, `open()` and `stat()`. A large view of a `read()` system call is shown in figure 2.9.

Scheduler I/O requests (3) Block requests are plotted as thin lines in the time line for the process responsible for the request. When dispatched from the kernel a line starts in the top, and upon completion the line ends. This means that the angle of each line shows clearly how much time was spent between dispatch and completion, and it also allows us to see when requests are delayed by the NCQ mechanism in the disk.

We have also divided requests into three classes and color coded them according to the *dispatch-to-completion* (D2C) time:

Cached requests are colored green, and are detected by a very low D2C time, below 1 ms. This is most likely due to readahead caching in the disk.

Sequential requests, 1 to 5 ms, are orange. This should cover transfer time and track to track seeks for most disks.

Seeks are colored red, over 5 ms

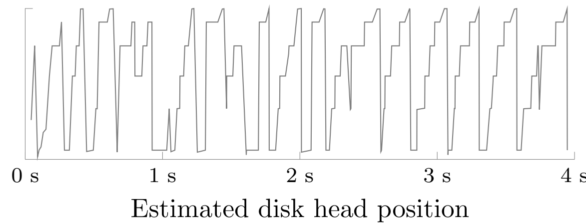


Figure 2.10: C-SCAN elevator visualized

These classifications are only meant for hints, and the classification only makes sense for rotating disks, not solid state disks. As there are often many requests with similar behavior, we have merged consecutive requests within the same class from the same process, which are then rendered as a parallelogram.

In figure 2.9, we have zoomed in on one `read()` request, and we can see how the process is blocked for a while before the requests are dispatched. We can see that once the requests have completed, the system call returns. We also see that one of the requests required a seek: it is highlighted in red and the angle indicates how long the disk used to process the request. In this case, it might indicate that the file was fragmented.

Disk head position (4) We plot the estimated disk head position by drawing lines between the sector offset for the disk completion events. It is an estimate, because we do not know exactly how the disk works, but we assume that for any logical sector address a and b,

$$a < b \Rightarrow \text{cylinder}(a) \leq \text{cylinder}(b)$$

As shown in figure 2.10, this gives a good indication on what kind of elevator algorithm, if any, is used. The bottom of the Y-axis indicates the outermost cylinder.

Bandwidth graph (5) The bandwidth graph shows the average throughput for the block device. When the scheduler supports bandwidth reservation/QoS, we have indicated the reserved bandwidth with a red line.

The Base Workload

We have created a mixed workload which highlights the behaviour of each scheduler. We want to evaluate the performance, latency and fairness of

Parameters		Performance		
Scheduler	NCQ	Bandwidth (KB/s)	Bandwidth %	Latency (random I/O)
cfq (long slices)	Off	65528	100	341
anticipatory	Off	62527	95.29	266
anticipatory	15	60803	92.66	413
anticipatory	31	59989	91.42	377
cfq	Off	56550	86.18	74
cfq	31	42654	65.00	101
cfq	15	40222	61.29	76
deadline	31	32069	48.87	91
deadline	15	31754	48.39	79
noop	15	31498	48.00	86
noop	31	31462	47.94	78
deadline	Off	23094	35.19	83
noop	Off	20795	31.69	92

Table 2.3: Results of mixed load on different scheduler configurations

each scheduler. The load includes high-bandwidth streaming requests (S_n), random requests (R_n), and a backup process reading multiple files (MF_n). The random requests are confined to one 8 GiB large "database file", while the backup process traverses a full directory tree and accesses data wherever the file system has allocated it.

Deadline I/O Scheduler Performance

The mixed load benchmark presented in figure 2.11 shows that throughput is low when multiple streaming requests ($S_0 - S_3$).The visualization of the estimated disk head position shows that the elevator works, requests are handled only as the elevator goes up (C-SCAN). However, in the same graph we can also see that the elevator does approximately 16 passes over the disk every second, which is very expensive. The reason for this is not directly apparent from the scheduling algorithm, but by looking at the I/O visualization, we can see that requests are not batched properly, the scheduler quickly switches between streams and does not handle one sequential `read()` request at a time.

Further inspection of the `blktrace` output, shown in listing 2.2, reveals the problem:

1. Process A queues up two sequential I/Os (`Q`)
2. The I/O scheduler dispatches the requests (`D`)
3. The request is completed and is returned to the FS (`C`)

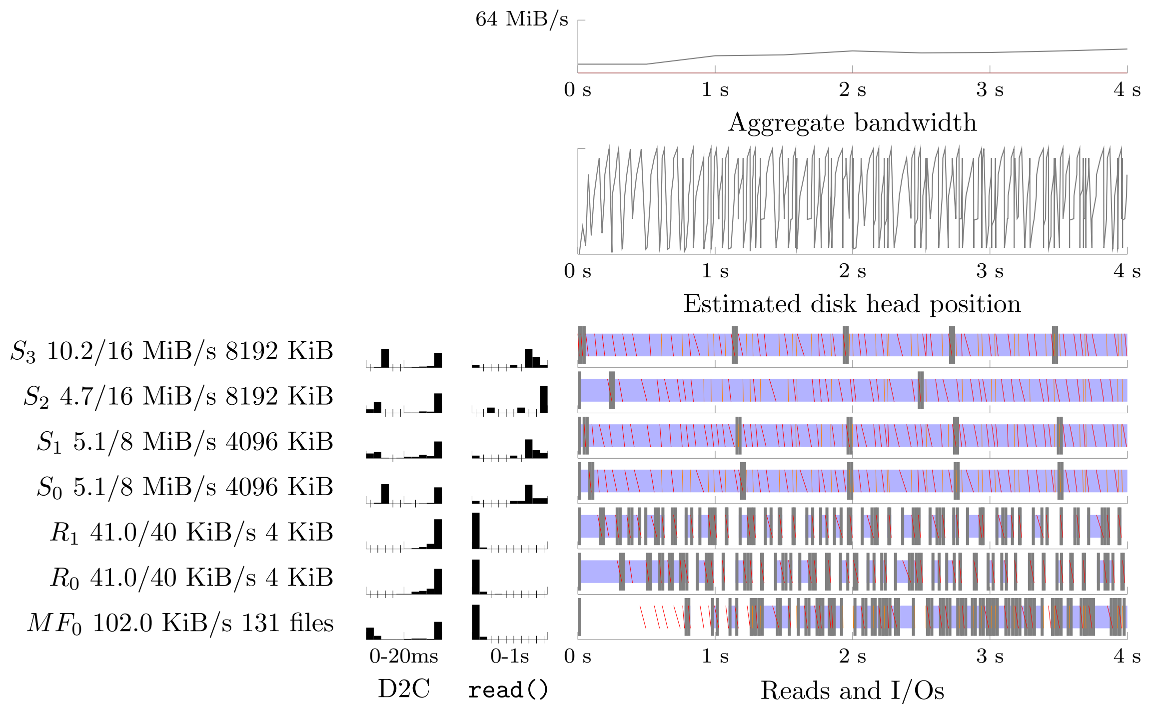


Figure 2.11: Deadline scheduler example

```
   Dev  CPU Seq   Time           Event Sector    + Size
1: 8,16 5   462   3.586650236    Q   R 90177087  + 256 [Process A]
   8,16 5   466   3.586724886    Q   R 90177343  + 256 [Process A]
[...]
2: 8,16 1   487   3.643737421    D   R 90177087  + 512
[...]
3: 8,16 5   469   3.646509764    C   R 90177087  + 512
4: 8,16 5   470   3.646582420    D   R 323194943  + 1024
5: 8,16 5   472   3.646662913    Q   R 90177599  + 256 [Process A]
   8,16 5   476   3.646757798    Q   R 90177855  + 256 [Process A]
```

Listing 2.2: Deadline `blktrace` output

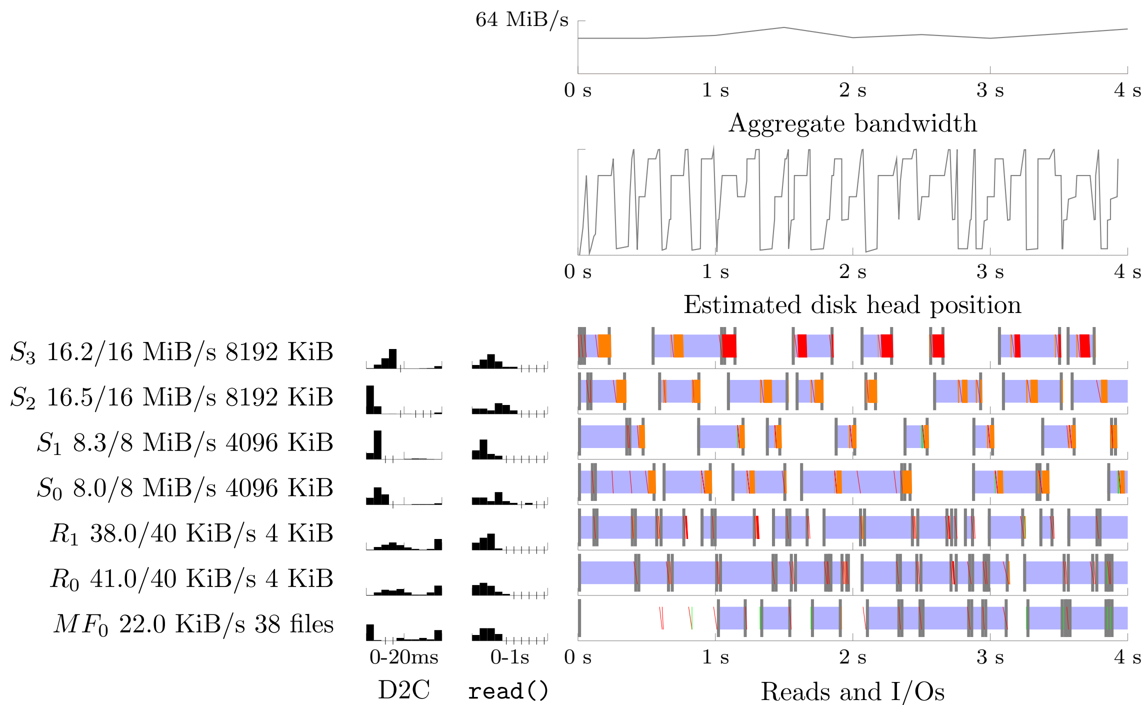


Figure 2.12: Anticipatory I/O Schedule example

4. The I/O scheduler is asked to dispatch another request, and picks the next request from the queue (`D`)

5. Process A queues up two new requests, sequential to the previously completed request (`Q`)

When the deadline scheduler chooses the next I/O request to serve in step 4, the file system has not yet queued up the next block in the `read()` operation for process A. 80 microseconds later, the file system queues up the request for the process, but at that time, it is too late, and the elevator has left. The Deadline I/O scheduler does not support backwards seeking at all, even though it is possible that the blocks succeeding the previous request are stored in the disk drive cache due to read ahead.

Anticipatory I/O Scheduler (AS) Performance

We see that the disk head position graph for the Anticipatory I/O Scheduler in figure 2.12 *looks* very nice, and indeed the bandwidth requirements of most streams are satisfied. This means that the anticipation algorithm pays off in this example. The streaming processes get a consistent and low

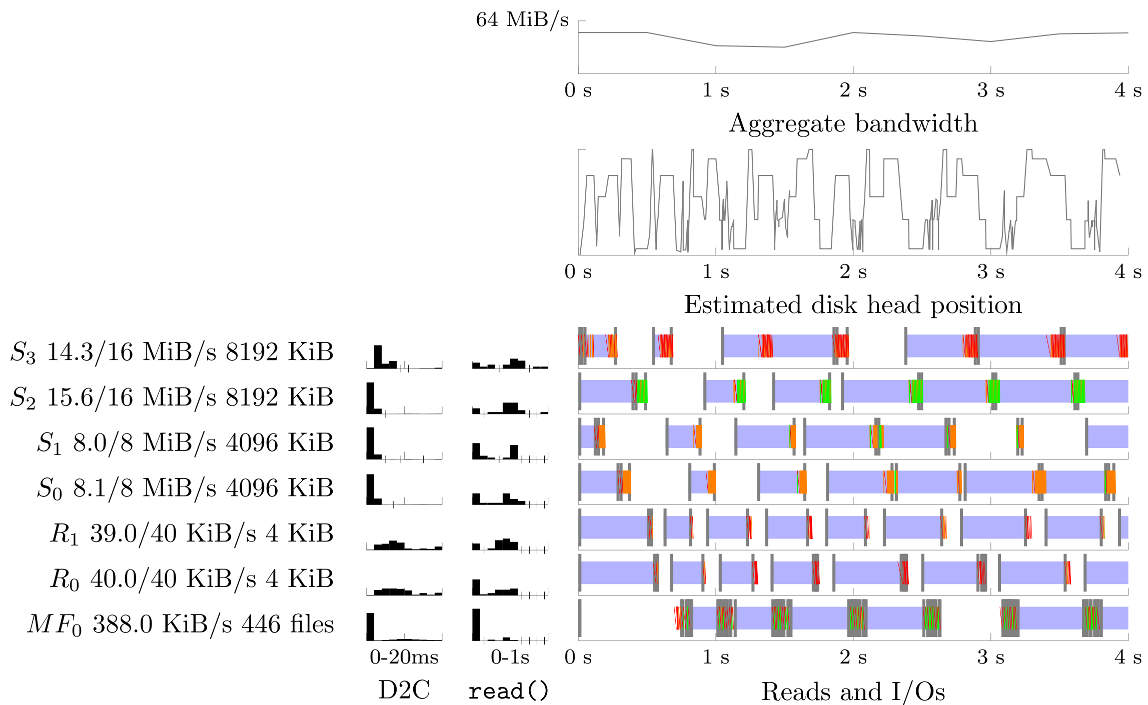


Figure 2.13: CFQ I/O Schedule example

`read()` time, and many `read()` system calls are serviced without interruption.

We can also see that the random reading processes, which are confined to 8 GiB files, sometimes are serviced multiple times in a row. This means that the anticipation logic considers it worth waiting for the process, because it has low inter-arrival times (low think time), and the 8 GiB large data file is small enough to consider it as short seeks (better than switching to another file).

The major problem here is that the multi-file I/O process MF_0 is almost starved. We can see that the process is allowed only *one to two I/Os* on each round of the elevator. This is because the file system has placed the files all around the file system, and the process accesses different types of data (file metadata, directory data and file data) which are spread around the disk. The process is only serviced when either the elevator passes by, or when a deadline has expired (more than 125 milliseconds have passed since the last request was queued).

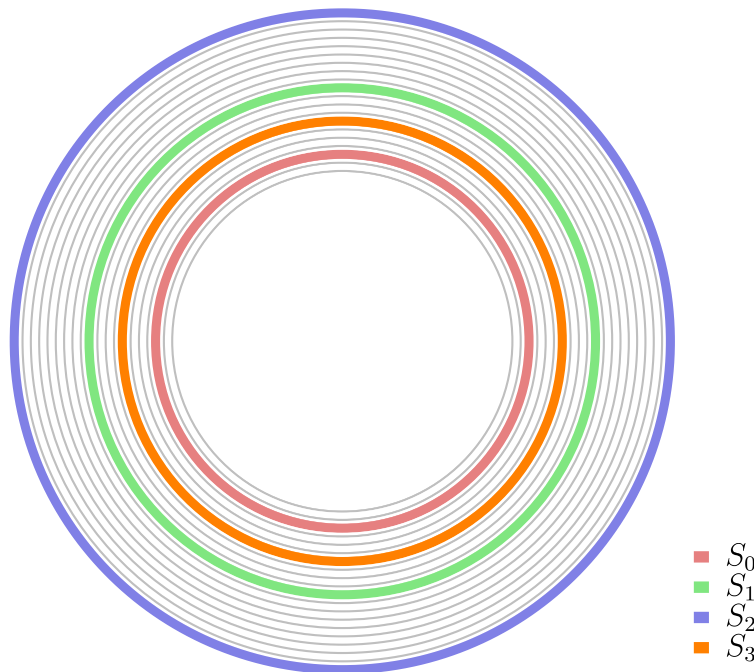


Figure 2.14: Estimated cylinders of streams

Complete Fairness Queueing (CFQ) Performance

The overall performance of CFQ is on par with what we found with the Anticipatory I/O Scheduler, but one difference is worth noting: The multi-file process (MF_0) got far more work done, on the expense of the two high-bandwidth streams (S_2 and S_3).

The definition of fairness in CFQ is that two processes with the same priority will get the same amount of disk time. It may then be a bit confusing to see the difference in throughput of two streaming processes, S_2 and S_3 in figure 2.13. S_3 gets much lower throughput—even though they are in the same priority class and request the same bandwidth. This is entirely due to zoning (shown in figure 2.3), the file used by S_3 is stored on one of the inner tracks of the disk. In figure 2.14, we have shown the estimated physical cylinder, by using the logical address of the first block in each file. This shows one of the issues with the *time based* QoS system in CFQ.

Another issue is that a *proportional share* scheduler does not give any absolute guarantees; when priority N might be satisfactory with three readers, it may turn out to be too little when another reader is added, and each proportion consequently shrinks. In figure 2.13 we can see that the default priority level was not enough for S_2 and S_3, they were unable to read 16

MiB/s. While giving a process the highest priority level (0) and/or using the RT class could work, we would then run into a new issue. If the process read data faster than expected, it would starve other processes and as a consequence they would miss their deadlines. It is difficult, if not impossible, to combine the existing CFQ RT class with *work conservation*. A process does not know whether it can read more data (to fill buffers) without disturbing other processes.

The disk seek graph in figure 2.13 shows that there are three issues preventing maximum global throughput:

Lack of ordering between processes The seek graph shows that there is no elevator on a larger level, the ordering of time slices is only due to I/O priority and is otherwise arbitrary. This may be considered a regression, as it was not an issue with the Anticipatory I/O Scheduler.

In figure 2.15, we have indicated how a high priority best effort task would be inserted (the black arrow) when the current time is 200. As we can see, this is only based on service time and class, the sector is not considered at all.

For schedules involving four or more processes, this adds unnecessary long seeks. With a schedule of only three processes, all the six possible schedules are already ordered as C-SCAN (but may be reverse). Figure 2.16 shows how the aggregate bandwidth varies with all possible permutations of 7 streams[4]. The best order (0, 1, 3, 4, 6, 5, 2–which is SCAN order) gives 71.8 MiB/s while the worst order (0, 6, 1, 4, 3, 2, 5) gives 67.8 MiB/s. Because CFQ does not have a top-level elevator, we cannot know which permutation of ordering we get. If, however, the scheduler had a top-level elevator doing C-SCAN ordering, we would probably get better global throughput. With variable round time it will not be possible to service all processes in one round at a time, but *some* reordering could be done in order to improve performance.

The lack of reordering might either be because it is assumed that the number of busy queues will most often be less than four, or that the gain shown in figure 2.16 is not worth it, considering the added complexity and response time jitter which reordering might create.

No collaboration With two or more processes seeking across the same area (with seeking I/O like the MF_0 process) global throughput could be

[4] There are $6!$ different permutations, because we have removed all permutations which are rotations of another permutation. E.g., (2, 3, 1) and (3, 1, 2) are rotations of (1, 2, 3)

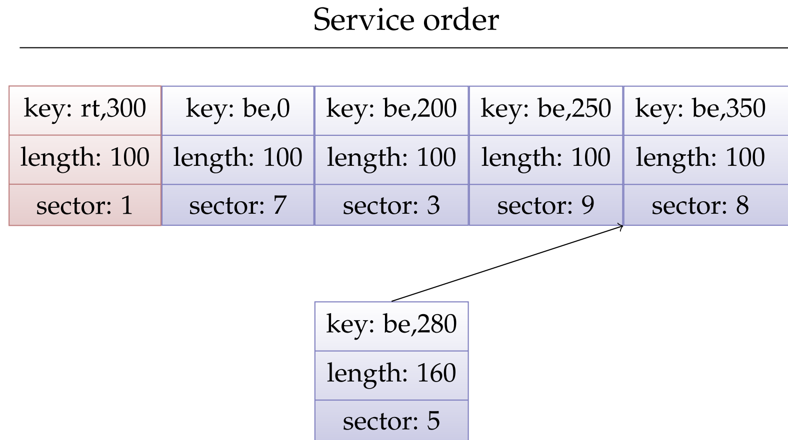


Figure 2.15: Insertion of a new best effort priority level 1 task into a CFQ Service Tree

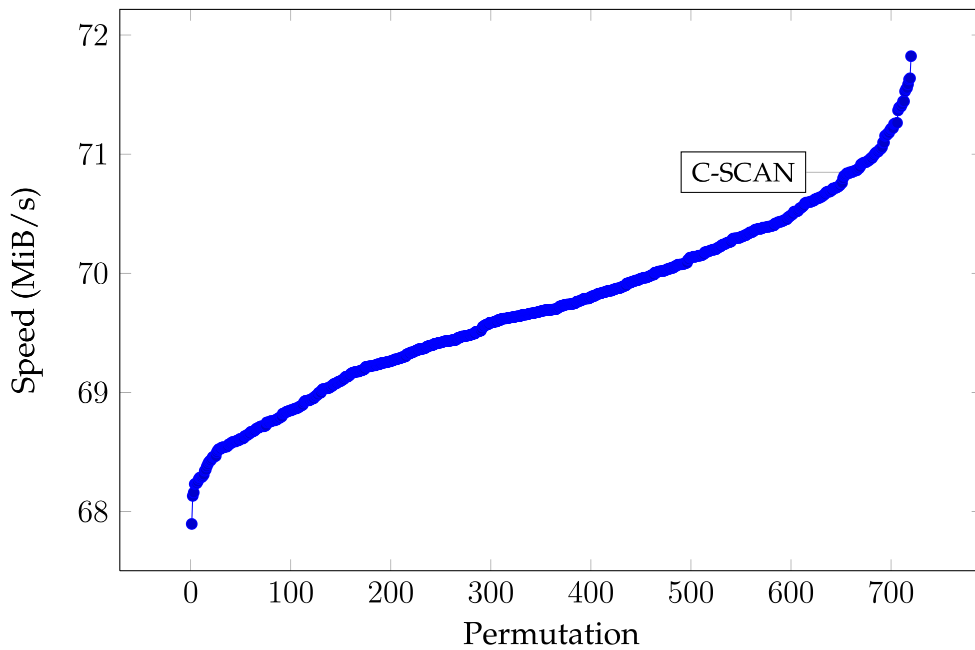
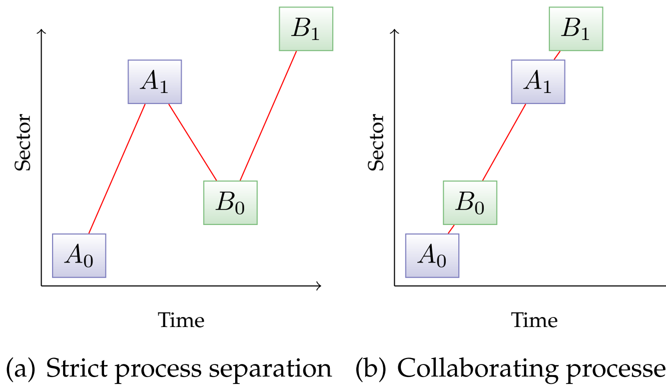


Figure 2.16: Performance variations with random ordering of 7 streams with 3 MiB reads



(a) Strict process separation (b) Collaborating processes

Figure 2.17: Collaborative I/O versus strict separation between processes

improved by interleaving requests for both processes in order to minimize total seek length. Currently, CFQ gives exclusive access to one process during a time slice. An illustration of this issue is shown in figure 2.17.

However, with realistic benchmarks based on the original work load we have not been able to show any significant loss of throughput: We let two `tar`-processes archive two different directories, in which the file data and inodes in both directories were interleaved. The fact that we did not find a significant difference between CFQ and the other schedulers may either mean that the collaboration effect is not achieved with the three other schedulers, that the test is not good enough, or that the gain of collaborative I/O is low.

With what we consider a very unrealistic scenario we did manage to provoke worst-case behaviour in CFQ, as shown in figure 2.18. We have N processes performing random reads alternating between the innermost and outermost cylinder, so with a good scheduler there will be an increase in I/Os per second for each additional process.

No re-ordering for the multi-file process An important observation is that the multi-file process does a lot of seeking, and the CFQ scheduler does not do anything about that. This issue is twofold:

- Because the scheduler uses time slices with exclusive access, any potential in improvement due to collaboration is not utilized, as we already explained in earlier in this section. Solving this is also difficult because we could end up with the same lack of fairness as in the other schedulers we have discussed.

- The process uses synchronous I/O, so there is never more than one

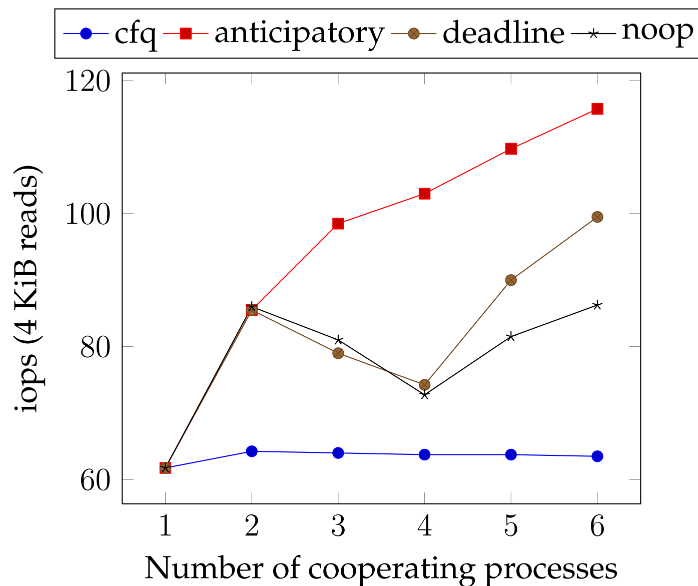


Figure 2.18: Collaborative I/O performance

pending request at a time. This means that *the scheduler cannot do any reordering*. In section 2.3 we will study the cause of the seeking, and a proposed solution is implemented and evaluated in chapter 4.

In chapter 3 we will try to solve the QoS and performance problems we have found with CFQ in a streaming scenario.

Related Work In this section, we give a short summary of *currently active* approaches to bandwidth management for the Linux kernel:

dm-ioband This is a bandwidth controller which is implemented above the I/O scheduler. Bandwidth allocation is proportional share, so like with CFQ there are no guarantees. The can only give her process a large relative priority and hope that it is enough. Dm-ioband is work-conserving, because tokens are refilled when all the processes with I/O pending have used all their tokens.

cgroup io-throttle As the name says, cgroup io-throttle is only *throttling* I/O, it does not reserve any bandwidth for your process. This could be worked around by assigning a low bandwidth to the default group, but this is not a sufficient solution. If a process in the default group is seeking, it will consume a lot of *disk time* without using up its tokens.

Io-throttle is implemented at a high level; inside the VFS and memory management code. Another problem is that the controller is not work-conserving, so the system will not be fully utilized. Recently a form of work-conservation was added, by disabling throttling as long as the disk utilization is below a certain level.

BFQ This is a modification of the CFQ I/O Scheduler that uses bandwidth instead of disk time [15]. We argue that both are needed, and that disk time is a better alternative for sharing the disk between multiple processes. Each process will have to be assigned a low bandwidth to prevent starvation if a process is seeking a lot. The problem with this is that sequential reading processes, such as media streams, are often preempted, and the global throughput suffers.

IO Controller This is another proportional weight framework, which uses the code from BFQ for fairness. It does not attempt to reserve bandwidth or support variable deadlines.

The general problem here is that these approaches do not consider media streams, and are mostly designed to solve problems with virtualization and that it is too easy to create a lot of asynchronous write operations, which also consume memory until written to disk. We would also argue that bandwidth control should be done inside the I/O scheduler for better efficiency. More information about this subject is found at [16].

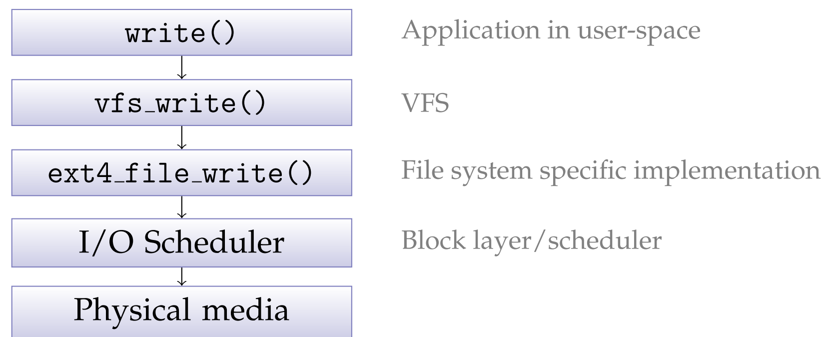


Figure 2.19: The position of the file system and VFS in the I/O path

2.3 The ext4 File System

In this section, we will study the on-disk layout of the ext4 file system[5], in particular we will focus on the disk access patterns when looking up files, reading meta-data and data. This is important because it affects I/O performance. We assume some knowledge of Unix file system concepts [17]. After reading this chapter you should have a good understanding of the ext4 file system layout. We have chosen the ext4 file system because it will most likely become the most widely deployed Linux file system when the next versions of GNU/Linux distributions are released. The structures and concepts introduced in this chapter will be the foundation for the techniques used in chapter 4.

2.3.1 The Virtual File System

The Virtual File System (VFS) is an abstraction layer inside the kernel that provides a common interface to the different file system implementations. The VFS is sometimes called the Virtual File Switch, and it allows multiple file systems to co-exist. In Unix, file systems are all sharing a common name space, so they all make part of the same tree structure, as opposed to in Microsoft Windows where each drive has its own root, like `C:` and `D:`.

When a user space application enters the `write()` method, which takes a file handle and some data as parameters, the generic function `vfs_write()`

[5]As ext4 is a highly backwards compatible file system, it has many feature flags which may be enabled and disabled. In this chapter, we assume that the default features `sparse_super`, `extents`, `dir_index`, and `flex_bg` are enabled, and that the file system was freshly created, not upgraded from ext2 or ext3. Some of our observations may also be specific to Linux 2.6.28.

Parameter	**Value**
Block size	4 KiB
Blocks per group	32768 (128 MiB)
Inodes per group	8192
Inode size	256 B
Flex group size	16 block groups (every 2 GiB)
Max extent size	2^{15} blocks (128 MiB)
Sparse block groups 50 GB FS	12
Sparse block groups 500 GB FS	17

Table 2.4: `mkfs.ext4` defaults in version 1.41.3

is executed. This function uses the file handle to invoke the file system specific file implementation of the `write()` method. In figure 2.19 we have shown the call path when the ext4 file system is used.

2.3.2 Disk Layout

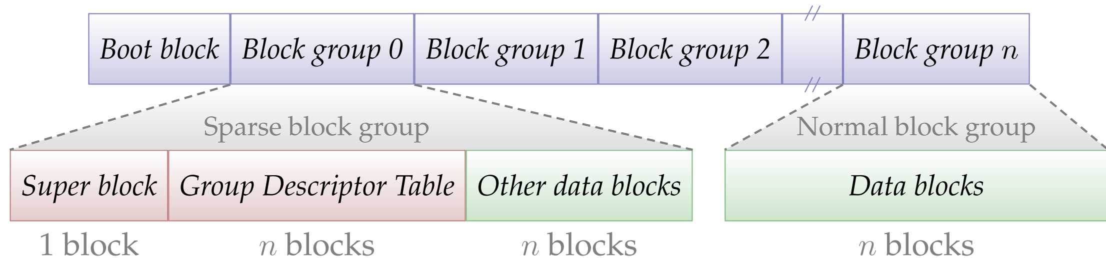


Figure 2.20: ext4 File System Layout

The ext4 file system divides the disk into a set of block groups, which are logical groups of sequential blocks [18]. Each group has the same size, which is determined during file system creation. The default values for the current version of the file system builder `mkfs.ext4` are shown in table 2.4.

Group Descriptors

Each block group is described by a Group Descriptor, stored in the Group Descriptor Table (GDT). A group descriptor contains pointers to data and inode allocation bitmaps, a pointer to an inode table and usage accounting for the bitmaps.

When the file system is mounted, the superblock and GDT are read from disk and stored in RAM. The original superblock and GDT are stored in the beginning of block group 0. Because the superblock and GDT data are crucial for file system recovery, they are checksummed and backups are stored in group 1 and groups that are powers of 3, 5, and 7. The block groups which contain backup data are called sparse groups.[6]

To allow file system expansion at a later time, the `mkfs.ext4` program reserves space for some additional Group Descriptors.

Inode and Block Bitmaps

To indicate whether a block or inode is free or allocated, the file system uses bitmaps. The bitmaps work as expected, e.g., bit 0, byte 0 is for block 0, bit 5 byte 0 is for block 5, bit 1 byte 1 is for block 9 etc. If the bit is set, the block is in use.

Each block group has one bitmap for inodes and another for data blocks, which are both one block in size. This puts an upper bound on the number of blocks in a block group to the number of bits in a block, which is $8 \cdot bytes_per_block$, i.e., 128 MiB for the default 4 KiB block size.

Virtual Block Groups (`flex_bg`)

For better I/O performance, the bitmaps and inode tables are collected together in every Nth block group, making virtual block groups which are N times as large. The default `flex_bg` size is 16, so block group 0 contains the bitmaps for group 0 to 15, and block group 16 contains the bitmaps for group 16 to 31. In many ways, setting `flex_bg` to N makes block groups N times as large, effectively lifting the performance limitations imposed by the many block groups needed for large file systems.

The meta-data blocks in the virtual block groups are clustered by type, so the N block bitmaps are stored first, then the inode bitmaps, and finally the inode tables.

2.3.3 Inodes

An inode data structure contains meta-data about a file or directory. The most important fields in the inode is owner, access permission, modification times and pointers to the data blocks. In ext4 the inodes may be 128

[6] In the original ext2 file system, all block groups were sparse groups. As file systems grew, the overhead became too much, and the `sparse_super` feature was added.

or 256 bytes large, the latter is default. The full data structure for ext4 on Linux is shown in listing 2.3.

The data blocks for storing the inode tables are pre-allocated at file system creation, and the number of inodes is fixed after that—unless the file system is re-sized. The default ratio is one inode for every 8 KiB of disk space, which should be more than enough for most uses.

Inodes are accessed by inode number. Given an inode number you can calculate the block where the inode is stored directly, without reading any other data from the disk (we need the GDT but it is always stored in RAM).

2.3.4 Directories

A directory links file names to inodes, so we may refer to files using paths instead of numbers.

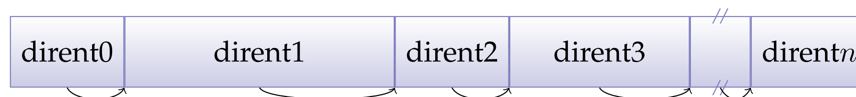


Figure 2.21: Variable sized directory entries on disk

Directories are stored as linked lists of directory entries, which is a data structure with inode number, name and type (see listing 2.4). The directories are linked lists because they are packed (the string `name` is variable length on disk), which prevents random access to the directory entries meaning that the list must be traversed and searched. The records are padded to multiples of four bytes.

When an entry is deleted the padding of the preceding entry is expanded to include the deleted entry. This can be seen in figure 2.22—where `file-4` has been deleted. Allocation of directory entries will be discussed in section 2.3.8.

2.3.5 Large, Indexed Directories

The directory layout described in the previous section is compact and efficient for small directories, but the $O(n)$ performance becomes unacceptable as the number of files within a directory grows. Directory indexes were introduced to solve this issue and give $O(\log n)$ performance. When the storage capacity for the directory entries in a given directory exceeds one block, the file system transparently switches to the HTree data structure [19], and sets the inode flag `EXT2_INDEX_FL`.

```
/*
 * Structure of an inode on the disk [simplified - Linux version]
 */
#define EXT4_N_BLOCKS 15
struct ext4_inode {
  __le16 i_mode;         // File mode
  __le16 i_uid;          // Low 16 bits of Owner Uid
  __le32 i_size_lo;      // Size in bytes
  __le32 i_atime;        // Access time
  __le32 i_ctime;        // Inode Change time
  __le32 i_mtime;        // Modification time
  __le32 i_dtime;        // Deletion Time
  __le16 i_gid;          // Low 16 bits of Group Id
  __le16 i_links_count;  // Links count
  __le32 i_blocks_lo;    // Blocks count
  __le32 i_flags;        // File flags
  __le32 l_i_version;
  __le32 i_block[EXT4_N_BLOCKS];// Pointers to blocks
  __le32 i_generation;   // File version (for NFS)
  __le32 i_file_acl_lo;  // File ACL
  __le32 i_size_high;
  __le32 i_obso_faddr;   // Obsoleted fragment address
  __le16 l_i_blocks_high; // were l_i_reserved1
  __le16 l_i_file_acl_high;
  __le16 l_i_uid_high;   // these 2 fields
  __le16 l_i_gid_high;   // were reserved2[0]
  __u32  l_i_reserved2;
  __le16 i_extra_isize;
  __le16 i_pad1;
  __le32 i_ctime_extra;  // extra Change time
  __le32 i_mtime_extra;  // extra Modification time
  __le32 i_atime_extra;  // extra Access time
  __le32 i_crtime;       // File Creation time
  __le32 i_crtime_extra; // extra FileCreationtime
  __le32 i_version_hi;   // high 32 bits for 64-bit version
};
```

Listing 2.3: Ext4 inode

```
#define EXT4_NAME_LEN 255

struct ext4_dir_entry_2 {
  __le32   inode;               /* Inode number */
  __le16   rec_len;             /* Directory entry length */
  __u8     name_len;            /* Name length */
  __u8     file_type;
  char     name[EXT4_NAME_LEN]; /* File name */
};
```

Listing 2.4: Ext4 directory entry

```
Hexadecimal values                                 String values
02 00 00 00 0c 00 01 02  2e 00 00 00 02 00 00 00  |       .         |
0c 00 02 02 2e 2e 00 00  0b 00 00 00 14 00 0a 02  |     ..          |
6c 6f 73 74 2b 66 6f 75  6e 64 00 00 0c 00 00 00  |lost+found       |
10 00 05 02 64 69 72 2d  31 00 00 00 0d 00 00 00  |    dir-1        |
10 00 05 02 64 69 72 2d  32 00 00 00 0e 00 00 00  |    dir-2        |
20 00 06 01 66 69 6c 65  2d 33 00 00 0f 00 00 00  |    file-3       |
10 00 06 01 66 69 6c 65  2d 34 00 00 10 00 00 00  |    file-4       |
94 0f 06 01 66 69 6c 65  2d 35 00 00 00 00 00 00  |    file-5       |
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |                 |
```

Figure 2.22: Packed directory entries on disk, colors indicates inodes, record lengths, name size, type, names, padding, and deleted.

The HTree is a BTree-like data-structure, currently limited to two levels. The tree is indexed by a 31 or 63-bit hash of the file name. There are three hash functions available as of Linux 2.6.28, which hash function to use is set in the superblock. The default hash function is half-MD4.

In the first data block of the directory file, we find the root of the HTree, which references the leaf blocks—they are also parts of the directory file. Each leaf block is responsible for a range of hashes. In figure 2.23, we can see that directory entries with hashes between 0x00000000 and 0x0784f05a are found in data block 1. The leaf block is identified by the 31 most significant bits of the first hash in the range, while the least significant bit is used to indicate a collision which has been split (in which case the next block range may also have to be examined).

Within the root block we find fixed size entries of 8 bytes; four byte hash and four byte file block number. The entries are sorted, so look-up may be done by binary search. At this point we know which data block

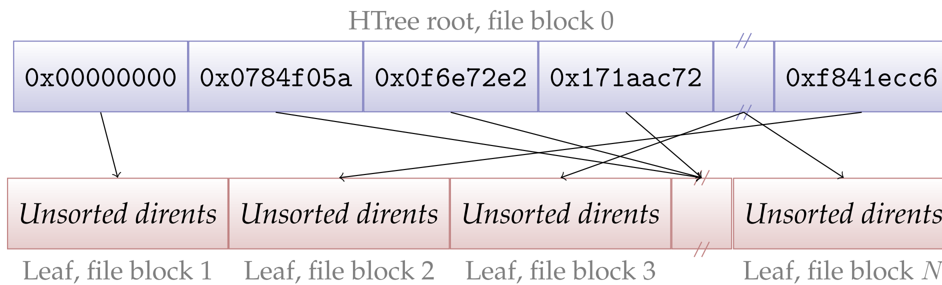


Figure 2.23: ext4 Directory HTree

contains the directory entry, so we need to read the directory block from disk and perform a normal linear search as in section 2.3.4.

The single level tree structure scales to directories with about 75 000 names in them—for larger directories a two level structure is used.

2.3.6 Name Look-up

The process of resolving a full path name to a file or directory, such as `/home/chlunde/master/main.tex`, is a recursive process. The VFS layer in Linux starts by looking up the inode of the root directory, `/`, which in ext4 has a fixed inode number of 2. From there on any path may be resolved by looking up the next name (first `home`) in the current directory file to get the inode number of the next component, until we have reached the last directory or file name.

For performance reasons Linux implements a cache for name look-ups, called the dentry cache (dcache). The cache uses a Least Recently Used (LRU) based replacement policy, it is file system independent and implemented in `fs/dcache.c`. A hash table is used for look-up via the `d_lookup` function.

All dentry objects in the dcache keep a reference to the backing inode, which effectively forces it to be kept in memory. This means that if a path is in the dcache, the inodes will also be cached [20].

2.3.7 Extents

An extent is a range of up to 2^{15} contiguous blocks of file data, i.e., up to 128 MiB with 4KiB block size. Extents are used by default for all newly cre-

```
/*
 * This is the extent on-disk structure.
 * It's used at the bottom of the tree.
 */
struct ext4_extent {
  __le32 ee_block;    // first logical block extent covers
  __le16 ee_len;      // number of blocks covered by extent
  __le16 ee_start_hi; // high 16 bits of physical block
  __le32 ee_start_lo; // low 32 bits of physical block
};

/*
 * This is index on-disk structure.
 * It's used at all the levels except the bottom.
 */
struct ext4_extent_idx {
  __le32 ei_block;    // index covers logical blocks from 'block'
  __le32 ei_leaf_lo;  // pointer to the physical block of the next
                      // level. leaf or next index could be there
  __le16 ei_leaf_hi;  // high 16 bits of physical block
  __u16  ei_unused;
};

/*
 * Each block (leaves and indexes), even inode-stored has header.
 */
struct ext4_extent_header {
  __le16 eh_magic;      // probably will support different formats
  __le16 eh_entries;    // number of valid entries
  __le16 eh_max;        // capacity of store in entries
  __le16 eh_depth;      // has tree real underlying blocks?
  __le32 eh_generation; // generation of the tree
};
```

Listing 2.5: Ext4 extent data structures

ated data files and directories on ext4[7]. When the inode flag `EXT4_EXTENTS_FL` is set, the head of the 60 byte field `i_blocks` in the inode will be interpreted as an `ext4_extent_header`, as defined in listing 2.5.

The extents are stored in a variable depth three, in which the depth of the current node is stored in the header. For small files, a list of extents will be stored directly in the inode structure as shown in figure 2.24. In this case we have a leaf node where `depth` will be 0, and the 60 bytes allow us to

[7] As a curiosity, we may note that we have observed that the root directory—which is created by `mkfs.ext4`, not the kernel file system driver—does not use extents but the old block pointer structure. Later versions of `mkfs.ext4` may change this.

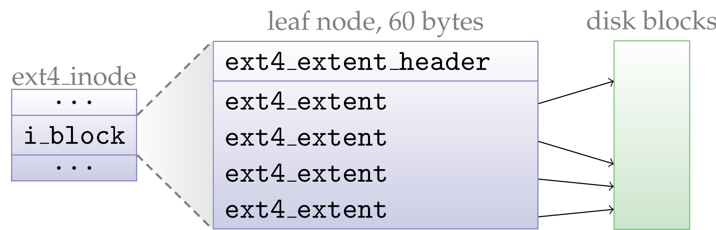


Figure 2.24: Extent tree with `depth=0`

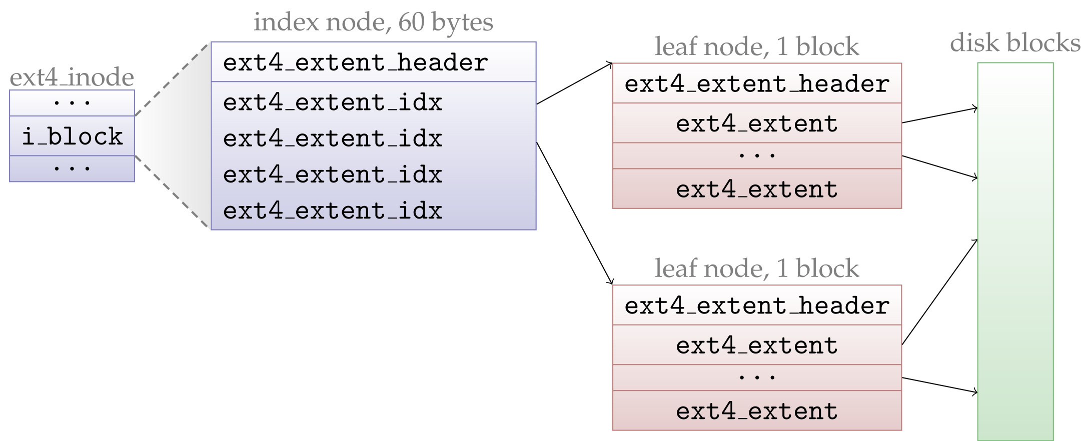


Figure 2.25: Extent tree with `depth=1`

store an array of up to four `ext4_extents`. This makes direct addressing for files up to 512 MiB possible—provided that contiguous space is available and that the files are non-sparse.

For large, sparse, or fragmented files more extents will be required. To support this, intermediate index nodes are provided, which are indicated by having a `depth` deeper than 0 in the header. Inside the inode we may then have four `ext4_extent_idx` entries. They will point to physical blocks, either leaves with a table of `ext4_extents` as in figure 2.25, or another level of intermediate node with a table of `ext4_extent_idx` for even more extents.

While the 60 bytes in the inode can only hold four `ext4_extent_idxs` or four `ext4_extents`, a full 4KiB block may hold 340. This means a single leaf node may address up to $340 \cdot 128$ MiB $= 42.5$ GiB of data, and one level extent tree with all four leaves in use can address at most 170 GiB. While a fully allocated two level extent tree in theory could point to 56 TiB of data, the actual maximum file size with 4 KiB blocks is 16 TiB, this is due to the

logical block pointers which are 32 bits.

2.3.8 Allocation

In this section we will look into the allocation algorithms used in ext4. Because the inode and extent allocation algorithms are subject to change, somewhat complex, and already well documented in [21], we will not go into every detail of their behaviour.

Inode Allocation

Ext4 has a new inode allocation policy to exploit the larger virtual block groups provided by the `flex_bg` feature. This is implemented by the function `find_group_flex` in `ialloc.c`.

The allocator first checks whether the virtual block group of the parent directory of the new inode has the following properties:

- It contains at least one free slot in the inode table
- It has more than 10% free data blocks

This results in good locality for directory tree traversal, and also increases the probability that the data blocks for the inode are placed close by, both for this and existing inodes. It allows for good locality if any of the inodes grow at a later point.

If this search is unsuccessful, the algorithm checks if the preceding virtual block group matches, hoping that maybe some data have been freed from that group.

The third option is to pick the lowest numbered block group on the disk matching the properties. Finally, as a last option, the block group with the largest free block ratio is picked.

Extent Allocation

The extent allocator has two policies, depending on the current file size and allocation request size. Both policies take a goal block as a hint, and return blocks from pools of pre-allocated blocks in memory. The goal block is located by `ext4_ext_find_goal`, which first tries to set the goal close to an existing data block which is logically near in the file. If there are no such blocks, the physical block of the best metadata structure is chosen, either the inode itself or the leaf of the extent tree in which the block would be inserted.

In the first policy, used when the files are considered small, the data blocks are taken from a per-CPU pool. The main purpose of this strategy is to keep smaller files close together on disk.

When the existing file size plus the allocation request exceed a tunable threshold value, the large file allocator comes into play. The threshold, which is compared to the current file size plus the new allocation request, is tunable in `/proc/fs/ext4/<partition>/stream_req`, the default value is 16 blocks. The large file allocator pre-allocates blocks per inode instead of per-CPU, which avoids large file fragmentation.

Buddy Cache If either of the pools are empty, a buddy cache is used to fill the appropriate pre-allocation space. The buddy cache is like the buddy memory allocator [22], i.e., all requests are rounded up to a power of two size on allocation. Then, the algorithm searches for a fitting extent, or performs a split if necessary.

There are no buddy allocator data structures on disk—they are built using the block bitmaps when needed and stored using a virtual inode.

For smaller allocation requests, the lower limit of pre-allocation is set by either the `/proc` tunable `group_prealloc` (defaults to 512 blocks) or the RAID stripe size if configured during mount (`mount -O strip=<size>`).

Delayed allocation Just like the write requests from the file system to the block layer, block allocation requests from the VFS to the file system also come one block at a time when using buffered I/O, even if the user space process issues a multi-block `write`.

With delayed allocation, the file system merely reserves space. It does not allocate any physical blocks when the VFS requests data blocks. Later, when the pages are flushed to disk, pages with no physical mapping are clustered and then allocated and submitted to the block layer for disk write.

User space initiated allocation To avoid fragmentation, user space applications should take advantage of the `posix_fallocate` function, which allocates data blocks in the specified range of the given file. Upon successful return, subsequent calls to `write` in the allocated space are guaranteed not to fail due to lack of disk space.

In the ext4 file system, this library function is backed by the system call `sys_fallocate`, which allocates large contiguous extents[8], and the data

[8] the file must be extents based

blocks are not initialized directly but just marked as uninitialized (the highest bit in the extent length field is set).

On the ext3 file system, the library method `posix_fallocate` behaviour is often undesirable, because it simply writes blocks of null-bytes to reserve storage by using `pwrite`. This wastes resources, the disk must write useless data, and the task doing the reservation is blocked during the writing. As the implementation is generic and not done in the file system, there is really no guarantee that fragmentation will be avoided by using the system calls.

Dirent Allocation

```
Hexadecimal values                                 String values
02 00 00 00 0c 00 01 02  2e 00 00 00 02 00 00 00  |      .          |
0c 00 02 02 2e 2e 00 00  0b 00 00 00 14 00 0a 02  |     ..          |
6c 6f 73 74 2b 66 6f 75  6e 64 00 00 0c 00 00 00  |lost+found       |
10 00 05 02 64 69 72 2d  31 00 00 00 0d 00 00 00  |    dir-1        |
10 00 05 02 64 69 72 2d  32 00 00 00 0e 00 00 00  |    dir-2        |
10 00 06 01 66 69 6c 65  2d 33 00 00 21 00 00 00  |    file-3       |
10 00 06 01 66 69 6c 65  2d 36 00 00 10 00 00 00  |    file-6       |
10 00 06 01 66 69 6c 65  2d 35 00 00 20 00 00 00  |    file-5       |
84 03 0a 01 6c 6f 6e 67  6e 61 6d 65 2d 36 00 00  |    longname-6   |
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |                 |
```

Figure 2.26: Figure 2.22 after insertion of "`longname-6`" and then "`file-6`". Colors indicate inodes, record lengths, name size, type, names and padding.

When a name is linked to a file name, a new directory entry is allocated. The allocation of directory entries is rather simple, if the directory is indexed, the leaf block with the matching hash is located, otherwise block 0 in the directory file will be used.

A linear scan then locates the next free space (i.e., gray area in figure 2.22) large enough for the new directory entry. This means a file named `longname-6` would be appended after `file-5`, while a shorter file name `file-6` would be inserted in the seventh line in the figure, as we have shown in 2.26.

If the directory entry does not fit within the block a split will be performed, the directory entries are rehashed and divided in two equally sized lists. The directory index will be updated to refer to the new blocks.

If the directory index is full and there is only a single level, it will be converted to a two-level structure. Otherwise a new second level index is added.

No garbage collection or packing is performed except for after a split, which means there may have been enough room in the original leaf node, and that the new leaf nodes may be less than 50% full after the split.

Another issue with the lack of garbage collection is that a directory that was once large will not shrink.

2.3.9 Journaling

The concept of journaling in file systems, or Write Ahead Logging (WAL) as it is called in database systems, was introduced to improve speed and reliability during crash recovery. The journal feature of ext4 (introduced in ext3, see [23]) is a circular buffer in which disk writes are logged before they are written to the main file system. This allows recovery because at all times the set of blocks which might be inconsistent is known, and the blocks are stored in a consistent state; either the new version in the journal, or the old version on the main file system.

The journaling feature has a large impact on the patterns of disk writes as seen from the I/O scheduler. While full data journaling increases the amount of writes by a factor of two, because data is written to both the journal and the main file system[9], the write patterns become much nicer.

The writes to the data journal, which may be either a pre-allocated file (represented using an inode number stored in the superblock–normally and extents based file with inode number 8), or a separate block device, are very sequential due to the nature of the log (append-only circular buffer).

Unless the file system is very small or a custom block or journal size is used, the journal will be 128 MiB large and stored in the middle of the file system, in a non-sparse block group without any inode tables and bitmaps.

After data is written to the journal, the file system may delay writing the data to the main file system longer than it could have done without journaling, especially when system calls such as `fsync` are used.

As shown in [24], the impact of journaling depends on the workload. For sequential writes, full data journaling may halve the bandwidth, while metadata journaling has low cost. On the other hand, with highly random

[9]It should be noted that it is possible to journal only metadata operations through a mount option

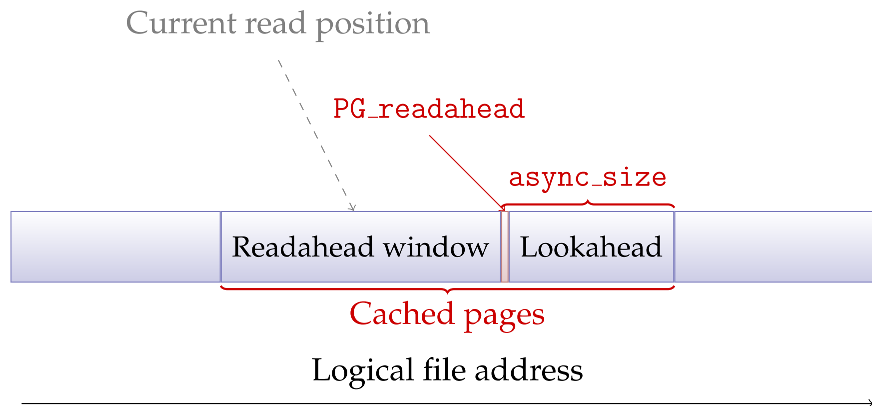


Figure 2.27: Readahead state

workloads data journaling may give a factor of 10 in performance gain, while plain metadata journaling again has little to no overhead.

2.3.10 Readahead and Caching

The basic idea behind readahead is to speculatively read blocks of data before the application requests them, so the application does not have to wait for the time consuming task of fetching data from the disk.

File Data Readahead

For readahead to be effective, there must be either predictability in the I/O patterns from a file, explicit hinting from the application, or even explicit readahead. The Linux kernel has a file system independent implementation of file data readahead for buffered I/O in `mm/readahead.c`. "On-demand readahead" [25] as the current algorithm is called, was introduced in kernel version 2.6.23.

The maximum benefit of readahead is when full pipelining is achieved: data is always ready before the application needs it. The on-demand readahead algorithm tries to achieve this by creating a lookahead-window after the readahead window. The last page in the readahead window is flagged with `PG_readahead`. When the flagged page is accessed through the page cache, the window ranges will be updated. This ensures there is data to process while fetching new data from disk.

In each `struct file` there is accounting data for the readahead algorithm, as show in listing 2.6. An application may influence the behaviour of the readahead algorithm by explicitly hinting what kind of I/O pattern

```
/*
 * Track a single file's readahead state
 */
struct file_ra_state {
  pgoff_t start;             /* where readahead started */
  unsigned int size;         /* # of readahead pages */
  unsigned int async_size;   /* do asynchronous readahead when
                                there are only # of pages ahead */

  unsigned int ra_pages;     /* Maximum readahead window */
  int mmap_miss;             /* Cache miss stat for mmap accesses */
  loff_t prev_pos;           /* Cache last read() position */
};
```

Listing 2.6: Readahead-state in file handle

it has. The system call `posix_fadvise` is used for declaring the file access pattern. Three of the flags are of interest for us here:

POSIX_FADV_NORMAL The default behaviour. Heuristics will determine if readahead is to be done.

POSIX_FADV_SEQUENTIAL The application notifies that accesses are expected to be sequential, so the Linux kernel doubles the maximum readahead size.

POSIX_FADV_RANDOM Accesses are expected to be random so readahead will be disabled.

For explicit readahead, an application may either use the system call `readahead`, or the flag `POSIX_FADV_WILLNEED` to `posix_fadvice`. Both operations initiate asynchronous I/O of pages in the requested range to populate the page cache.

Inode Readahead

In section 2.3.5, we explained that directory entries are stored in a random order when there are many files in the same directory. One problem with this is that if one wants to read all the inodes which the directory holds, and read them in the order returned from the file system, the disk will have to do a lot of seeking. Inode readahead tries to mitigate this issue.

When an inode needs to be fetched from disk the ext4 file system does not only read the block which contains the inode, but it also issues readahead requests for a larger part of the inode table. This is a spatial locality

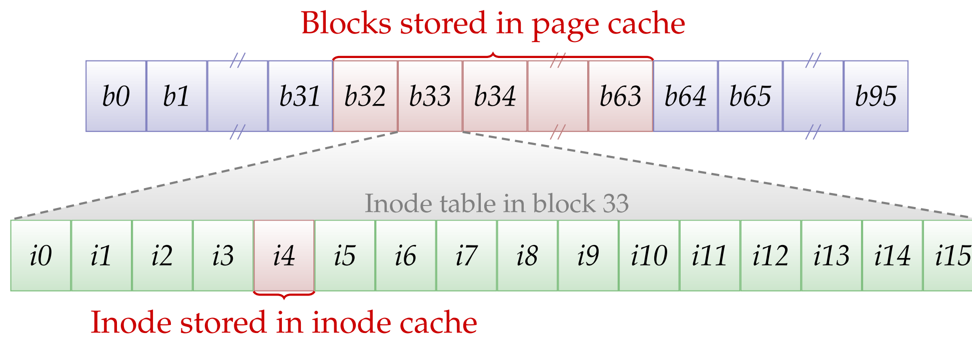


Figure 2.28: ext4 inode readahead

speculation from the file system, and if correct, the latency for nearby inodes is greatly reduced. Recall from section 2.3.8 that the inode allocation strategy is to place inodes close to the parent inode. The latency penalty for the larger read request is is relatively low. Another consideration is that we most likely will have to flush some other pages from the page cache to store the new data.

In figure 2.28 we show what happens when the fifth inode stored in block 33 is needed. The file system requests 32 blocks from the disk, aligned to 32. The blocks will be placed in the page cache, while the inode, once read from disk, will be placed in the inode cache.

The number of blocks to be read may be tuned at mount time by setting the mount option `inode_readahead_blks`, or at any later time by using the `/proc` file with the same name. The default value is 32 blocks, which on a standard file system means that blocks for $4096/256 \cdot 32 = 512$ inodes will be cached.

Only the requested inode will be parsed and stored in the inode cache, the other blocks will end up in the page cache and are parsed only if requested at a later time.

2.3.11 Implications of the File System Design

In this section, we will give a short summary of how the ext4 file system design will affect the access pattern of an application, and we will focus on applications reading multiple files. The issue with these applications is that they require seeking because of how the file system has allocated the data, and the I/O scheduler cannot reorder the I/O, because such applications only have one pending request at a time. Examples of such applications include `zip`, recursive copy and text indexing.

Metadata / Inodes

Small, new directories will often have files ordered by inode number, and the data is likely to have the same order as the inodes. However, when the directory grows and needs more than one block, the order effectively becomes random due to the hashing (see section 2.3.5). Within one file system block there could be a better ordering, but currently it is arbitrary. Even smaller directories have this issue, because when files are deleted and new are created, the listing will be unordered. As a consequence, reading the metadata for all files in a directory often becomes more expensive over time. The ext4 file system has introduced "inode readahead" to mitigate this issue somewhat, but it depends on spatial locality for related inodes, and with time it is likely that this property is reduced. Therefore, reading all metadata in a directory tree will likely be an operation requiring multiple disk seeks.

File Data Placement

In section 2.3.8, we explained how data blocks for new files were placed close to the corresponding inode, which again is allocated close to the parent directory inode, as long as there is room for data in that virtual block group. However, when the space in that block group has been used, another block group is used. This means that we may have to seek back and forth between block groups when we have new and old files in a directory. Another issue is that when we read multiple files, we jump back and forth between file data and meta data for each file we read.

Evaluation of Data Ordering Cost

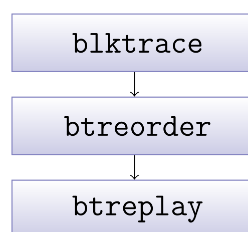


Figure 2.29: Trace replay data flow

In this section, we will try to quantify the cost of the order in which a program issues I/O. This will show us if there is any potential in optimizing a program by reordering requests. To do this, we have developed

a tool which couples with `blktrace` (the I/O tracing tool introduced in section 2.2.3). `blktrace` records all data I/O requests, so we can use the output to evaluate the performance of alternative orderings of the same requests. We use the existing `btreplay` tool to replay the trace, as shown in figure 2.29.

Our cost evaluation tool reorders the I/O request by sector number, in order to minimize seeks. We will define "best case performance" as the same I/O operations as the current implementation requires, done without any inter-request latency and dispatched sorted by block number. Because we run the trace on the same disk, with data placed by the file system, we get a more accurate result than if we had used a pure software simulation.

Because there often will be some data dependencies and processing time required, it will probably not be possible to achieve the performance of all data read with one arm stroke, but it will still be a good indicator of whether optimizing the I/O pattern of the system is worthwhile.

To simulate data dependencies, a situation where some data must be read before other data, we allow the evaluated software to inject barriers into the trace. Barriers are inserted by writing the literal string "barrier" to `/sys/kernel/debug/block/<blockdev>/msg`. Our reordering-program considers the barriers as ordering constraints: all requests logged before the barrier must be executed before the requests found after the barrier.

Implementation and Usage The evaluation method requires the user to benchmark the evaluated software while running `blktrace` to record all I/O operations. Then the user can run our reordering-program to create a new, sorted trace. To avoid data corruption, our program will issue write operations as reads, which tend to be somewhat faster than writes. Because we want to define an *upper* limit of the performance, this is not an issue. The sorted trace is replayed using the `btreplay` program.

Evaluating the `tar` Program In figure 2.30, we have shown the execution time of the unmodified GNU Tar program, the reordered trace, and how long it would take to read the same amount of data if it was allocated as one file. The result shows us that we can remove up to 35% of the runtime by reordering the I/O for the particular directory we archived (250 files spread over 8 subdirectories on an ext4 file system). We expect the improvement factor to increase as the number of files increases, and decrease for smaller file sets. The actual possible improvement is less than this, because we did not put any barriers in the trace, and there are data

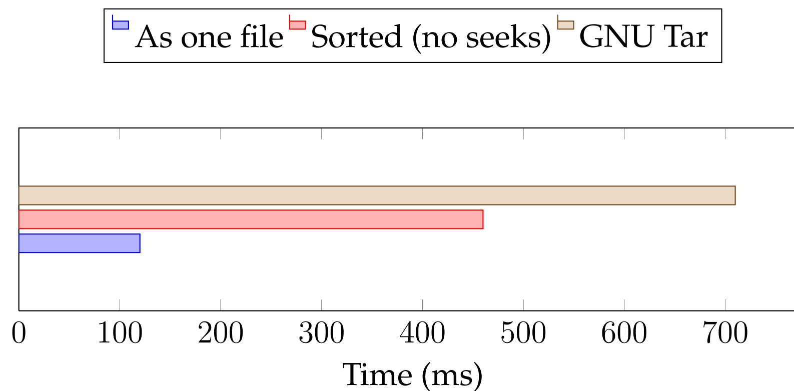


Figure 2.30: Potential in improvement for 250 files, total 7.88 MiB

dependencies (directories must be read to find new files, inodes must be read to find the file data location). Additionally, the CPU time of the application is not considered. Although some of the problems discussed here have been ext4 specific, such as hashing, the general concept should be valid for most file systems with directories and files.

The number for "As one file" shows us how long it would take to read the data if it was done as one sequential read of a file without any gaps. We cannot reach this performance without modifying the allocator algorithms in the file system, which we consider out of scope for this thesis.

2.4 Summary

We have shown how the I/O schedulers have evolved in Linux, and that there is currently no satisfactory QoS system for handling concurrent media streams with bandwidth and latency requirements. We have also shown how the file system stores data, so we know what data needs to be read in order to fetch metadata and file data. While the file allocator may try to keep related information close together, it cannot predict all usage cases, consequently there is bound to be seeking. In the next chapter, we will implement I/O scheduler changes in order to improve performance and QoS support, and in chapter 4, we will look at how applications can solve many of the seeking issues the file system creates—and which the scheduler cannot do anything about.

Chapter 3

Adding QoS to CFQ

In this chapter, we will design, implement and evaluate a QoS class for the CFQ I/O scheduler. Our main concern with the CFQ I/O scheduler is that proportional share is not a good match for multimedia with hard requirements for latency and bandwidth. We propose a solution to this by adding a new priority class to CFQ.

The main problem with the existing solutions is that they are often special purpose schedulers, which are not suited for server consolidation, where one wants to mix proportional share with fixed rate and deadline requirements. There are no available solutions included in Linux 2.6.29. There are several projects working on similar problems, but they are mostly for virtualization solutions, and instead of guaranteeing at least a certain bandwidth, they throttle to a bandwidth, without any work-conservation. This would mean that we would have to configure throttling of *all* processes if we wanted to guarantee a bandwidth for a single process. We do not consider this desirable on a multimedia/content server.

3.1 Design

With this in mind, we design a new QoS class for the CFQ I/O Scheduler, which we call Bandwidth (BW), which reserves bandwidth in terms of bytes per second in contrast to proportional disk time. The solution has the following design requirements and properties:

3.1.1 Non-intrusive

The implementation should not change the behaviour of the existing RT, BE and Idle classes of the scheduler. If there are no active BW class streams,

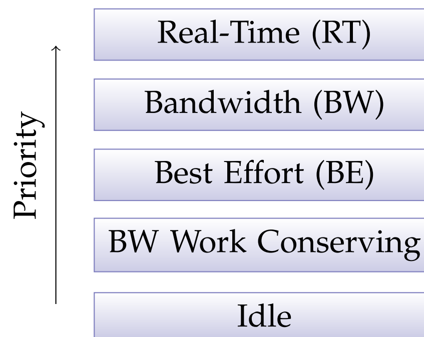


Figure 3.1: Priority levels in modified CFQ

the system should work as it does today. The class should be placed between the RT and BE classes, any real-time requests should be able to preempt the BW class.

3.1.2 Bandwidth (QoS)

An application must be able to request a specific bandwidth, and this bandwidth should be delivered to the application at the expense of any BE-class readers. The bandwidth should be measured by throughput as bytes per second, not with disk time and not proportional to load. In general, a process should not disturb other processes if it tries to use more bandwidth than it has reserved, and a BW process within its reserved bandwidth should not be affected by any BE process.

3.1.3 Request Deadlines (QoS)

To control buffer requirements in the client, an individual process may specify a deadline for its requests. The deadlines will only be soft, for two reasons: We do not implement proper admission control in the I/O scheduler, and we do not estimate the cost of I/O operations. There are still several advantages over the deadline support in the existing Linux I/O schedulers, which will make this solution far better for multimedia.

Deadlines in best effort CFQ are only relative to other requests the process has made, they do not affect when a process' queue is scheduled. In the Anticipatory I/O Scheduler and Deadline I/O Scheduler, deadlines are attached to requests from the VFS when they are queued. This means that if a `read()` has been split in two separate requests, the requests may be assigned two different deadlines. This is a problem if for example a `read()` operation needs to fetch an extent map block:

Consider an example where the scheduler operates with a 200 ms deadline, and we ask for 1 MB of data, which crosses boundaries of two extents. The first half, including the extent map, may be scheduled, e.g., after 150 ms. When the extent map is returned, a new request will be sent to the scheduler, with a new deadline of 200 ms, effectively setting a 350 ms deadline in this example. To avoid this, we do not assign a new deadline for consecutive requests with little idle time in between (at most 2 ms). This allows the VFS to parse the extent map and dispatch the new request.

3.1.4 Work-conservation

Any reserved bandwidth which is unused should be redistributed fairly amongst BE class readers, because it is important to support work-conservation to maximize global throughput. If no BE-class readers have requests queued, the BW-class readers who have exceeded their reserved bandwidth, but have pending requests, should be able to consume all the available bandwidth, as shown in figure 3.1.

There might be a number of reasons why a stream may reserve more bandwidth than it requests, for example Variable Bit-Rate (VBR) video and buffering. Conversely, VBR and caching might mean that the consumed bandwidth is less than the reserved bandwidth.

For simplicity, we only enter work-conservation in the BW class when there are no BE class readers with requests pending. A more complex implementation might treat a BW reader as a BE reader when they have exceeded the reserved bandwidth. We have not defined a policy for fair sharing of the extra bandwidth between BW class readers in the current design.

3.1.5 Global Throughput

To the extent allowed by the request deadlines and other requirements of the system, the scheduler should optimize the requests for higher global throughput by using an elevator.

3.1.6 Admission Control

A key component for getting a QoS system to work is admission control, a component which receives the QoS-requirements from an application (and characteristics about the round time, which file is going to be read,

etc), and judges whether the I/O scheduler will be able to serve the stream within the parameters. If we grant all requests for bandwidth and low latency, the promises will be broken.

Because we allow the RT class to preempt BW queues, an admission control system must also control RT class streams. To limit the scope of this thesis, we will not implement any admission control, we think this should be implemented outside the kernel I/O scheduler. There are several reasons for implementing the admission control in user space. For example, the admission control component could keep information about the performance properties of the disks in the system, and also know where files are positioned to take zoning into account.

A simple solution would be a daemon running as root, which an unprivileged process could contact using Inter-Process Communication (IPC) when it wants to reserve bandwidth. If the request is granted, the `ioprio_set` system call would allow the daemon to change the I/O priority of the requesting process.

3.2 Implementation

In this section, we will describe how we have chosen to implement the requirements and properties outlined from the "Design" section.

3.2.1 Bandwidth Management

Managing bandwidth with token buckets is a normal strategy for QoS and data rate control in networking systems. It has also been used before in disk scheduling, such as in the APEX I/O Scheduler [26]. Token bucket allows bursts, which are needed for good performance in a streaming scenario. If we instead used the leaky bucket, it is likely that we would run into the same seeking problem [27] as we showed with the Deadline I/O Scheduler. Even though bursts are allowed, we get good control over average bandwidth.

We have introduced four new variables to the per-process `cfq_queue` data structure. They are only valid if the process is in the bandwidth class:

`bw_last_update` The time, measured in internal clock ticks ("jiffies"), since we last gave tokens to this process. In the Linux kernel, the constant `HZ` defines how many clock ticks there are per second. On our system, this has a value of 1000, which allows for millisecond granularity.

bw_iorate The average rate, measured in bytes per second, at which this process is allowed to do I/O (unless we are in work-conservation mode). More specifically, it is the rate at which tokens are added to the bucket. This value is set by the user to control the bandwidth.

bw_tokens The number of tokens in the bucket. Each token represents one byte. We have chosen a signed integer for this value, so the number of tokens may become negative. This will be explained later. When a process queues its first I/O request, we fill the bucket with one second worth of tokens.

bw_bucket_size The maximum value of `bw_tokens`, i.e., how many tokens we may have in the bucket. This value is set by the user and controls the burstiness allowed. It is important that the round time is considered when setting the bucket size. If a process has a round time of two seconds, i.e., it issues a `read()` request every other second, the bucket size must minimum be twice the I/O rate.

bw_delay The maximum delay from a request has been queued until it *should* be dispatched, i.e., the deadline. This value is controlled by the user, who must also consider execution time of the request, and any other requests which might expire at the same time, when he/she wants to control the time each `read()` request may take. By setting a longer deadline the I/O scheduler with get more room for optimizing the I/O schedule, i.e., to dispatch requests in an order different from FIFO/Earliest Deadline First (EDF).

Management of the `bw_tokens` variable is fairly simple: When a request is dispatched, we remove the same amount of tokens from `bw_tokens` as the size of the request, converted to bytes. When we want to check if there are tokens available in the bucket, e.g., before dispatching a request, we must first check if any time has passed since we last updated the bucket. The new number of tokens is calculated as:

$$\texttt{bw_tokens} = \texttt{bw_tokens} + \frac{\texttt{bw_iorate} \cdot (\texttt{jiffies} - \texttt{bw_last_update})}{\texttt{HZ}}$$

However, if the number of tokens is larger than `bw_bucket_size`, we set the number of tokens to this value. If we are in work-conservation mode, the number of tokens may be negative. Consider a case where a process has been allowed to dispatch 40 MiB of I/O in one second, because no one else used the system. If the `iorate` of the process was only 1 MiB/s this would mean that the bucket would have -39 MiB tokens. If the system

then became 100% utilized with best effort processes we would not be allowed to schedule until 39 seconds have passed and there were tokens in the bucket again. While this is fair regarding the average bandwidth, 1 MiB/s, it would often be undesirable and could be considered unfair. To limit this effect we only allow the bucket to have $\frac{\texttt{bucket_size}}{2}$ negative tokens after work-conservation. By allowing the user to adjust the limit, this effect can be tuned, or completely disabled.

3.2.2 Deadline Support

For deadline support, we modify the original service tree, which we have named `deadline_service_tree`. The tree is sorted by queue type and deadline, which means that we can peek at the leftmost leaf to check if a deadline is reached. The first-level sorting (on classes) is as expected, RT, BW, BE, and lastly Idle, as shown in figure 3.1. The leftmost leaf of the deadline tree is checked to see if the either if a BW deadline has expired, or if there is an active RT queue. Either way this task will be serviced. In any other case, the sector sorted tree is used to pick the next queue.

We already mentioned the user-controllable per-process variable `bw_delay`. When a new request is dispatched, we use this value to insert the process into the deadline service tree. This is the weakest part of our implementation, we do not consider our neighbours when inserting a process into the deadline tree. If we have bad luck, there may be several processes with a deadline the same millisecond. This means deadlines are only best effort. We still argue that per-process deadlines in a higher priority QoS class are far better than the existing best effort alternatives.

An important consideration is to find a good deadline after a process has been in work-conservation mode (i.e., got higher bandwidth than requested). We already mentioned limiting the number of negative tokens to half a bucket. The deadline for a process with negative tokens is calculated as the time when the number of tokens becomes zero *plus* the `bw_delay` variable.

$$\texttt{deadline} = \texttt{jiffies} + \frac{-\texttt{bw_tokens} \cdot \texttt{HZ}}{\texttt{iorate}} + \texttt{bw_delay}$$

3.2.3 Inter-process Elevator

Because requests in the BW class will have known deadlines, we have more flexibility for the scheduler than the RT class. To increase global throughput, we try to minimize seeks by servicing requests in C-SCAN-order for as long as possible, until a deadline is reached. We introduce a

new data structure for each block device, which contains all active queues ordered by the lowest request in the queue. For performance reasons, we will use a Red-Black tree [7], a self-balancing binary tree with $O(\log n)$ look-up and insertion. We call this the `sort_service_tree`. We only insert BW class processes into this tree, as they are the only processes we are able to reorder without disturbing the fairness of the existing implementation.

3.2.4 Queue Selection

The queue selection algorithm runs whenever a new request may be dispatched to the underlying dispatch queue. Before doing so, a check is performed to see if the currently active queue should be expired or preempted. This will happen if any of the following conditions are true:

- The current queue is time sliced, and the time slice has expired
- The current queue is a BW class queue with a negative amount of tokens, and any of the following:
  - the next deadline is up
  - the next process is a BE process
- There is an active Real-Time queue (preemption)
- There is an active BW queue *with* tokens in its bucket (preemption)
- The current process has no I/O queued, and process history says that it is not likely to issue new non-seeky requests shortly, a small (typically 2-8 ms) delay is done to wait for new I/O. This is the anticipation implementation in CFQ, also known as the idle window. For BW class readers we have limited the anticipation to 2 ms because we expect only to wait for the VFS layer, not the process itself.

If none of these cases are true, we dispatch I/O for the current process or wait for it to queue new requests.

The queue selection (implemented in `cfq_set_active_queue`) is done in the following order:

1. The RT queue with the lowest deadline
2. The process at the head of the deadline service tree, if its scheduling deadline has expired. Note that in the presence of a BW class, the head will never be a best effort process, because they are ordered by class before deadline.

3. The BE class with the shortest deadline, if no BW class process has tokens left
4. The BW class in C-SCAN elevator order (or SSF, if configured to do so). The elevator starts wherever the disk head is assumed to be. This means that if an RT stream preempted a BW stream, and positioned the disk head in a completely different place, the elevator would continue in the new place, not the preempted stream.
5. The head of the deadline service tree

This order was chosen because it attempts to achieve all the qualities we specified in the design section.

3.3 Evaluation

We have done a performance evaluation of the new BW class compared to the existing solutions available in Linux 2.6.29. We have methodically designed scenarios to confirm or disprove that the implementation fulfils the design requirements.

To create a realistic multimedia scenario we consider a server which provides streaming video to multiple clients. High definition video (1080p) on the most popular storage format today (Blu-ray) has a theoretical maximum bitrate of 48 Mbit/s. We believe that the average bitrate for most high definition movies are around 25 Mbit/s (3 MiB/s), so we have chosen this bitrate for our video streams. To make sure the scheduler could handle different bandwidths, we added another class of video streams, CBR video at 1 MiB/s, which is at the upper side of DVD bandwidth.

In this section, we have used the word "greedy reader" for a process which consumes data at a rate faster than the disk can provide, for example the program `md5sum` consumed 325 MiB/s on our system, while the data rate of the disk is around 50-110 MiB/s.

3.3.1 Performance Limitations and Need for QoS

In the first scenario, we wanted to investigate what the maximum number of reserved readers is, that we could add while still maintaining deadlines. Each reserved reader had a deadline of one second for each request, so the scheduler had to serve each stream once per second.

We also wanted to evaluate how the priority methods available in the scheduler would allow us to run other jobs concurrently at best effort, so

we also added three processes doing random reads of 4 KiB, at most 20 times per second. Another set of best effort readers was added to see how streaming processes would impact the system performance and reserved readers, so we added five processes doing streaming at up to 4 MiB/s with a block size of 64 KiB. Note that we do not evaluate the performance of these streams: we are pushing the reservations to the maximum, with the consequence that they will be starved. If this is considered a problem, the admission control system must deny reservations before this happens.

Streams		CFQ			AS	Deadline	noop
1 MiB/s	3 MiB/s	BW SSF	BW C-SCAN	RT			
16	10	0	0	0	479	345	281
17	10	0	0	0	482	330	398
18	10	0	0	0	492	300	783
19	10	0	0	0	512	276	1061
20	10	0	0	179	505	288	1123
21	10	0	0	186	531	381	1075
22	10	0	0	276	517	947	1061
23	10	0	0	N/A	549	1233	1064
24	10	0	0	N/A	553	1276	1054
25	10	182	188	N/A	536	1279	1033

Table 3.1: Deadline misses with 1 s deadline / round

The results are shown in table 3.1. We started with 10 1080p-streams and 16 DVD streams, which all the CFQ-based solutions could handle, but the lack of any QoS in AS, Deadline and noop meant that they were unable to handle the RT streams any differently from the others, so the deadlines are missed. This shows that a priority mechanism *is needed and useful* in this scenario.

As we increased the number of DVD streams to 20, the RT class in CFQ failed to maintain the deadlines. We contribute this mainly to the fact that the RT class serves the streams in a FIFO-like order, without any high-level elevator. We can see that the BW class we created, which does reorder streams, can handle 5 more streams. In figure 3.4 we have shown how the elevators affect the disk head position. The conclusion is that it *is worth* adding re-ordering between streams, which in this case allowed us to have 17% more streams. The design goal of achieving higher global throughput has been achieved.

When the number of streams was increased to 33, we discovered a new problem with the RT class: one stream was completely starved (no I/O). This is the reason for “N/A” in the table. Similar behaviour occurs for BW but at a later time. Proper admission control should, however, never allow this to happen, so we do not consider this a problem with the schedulers.

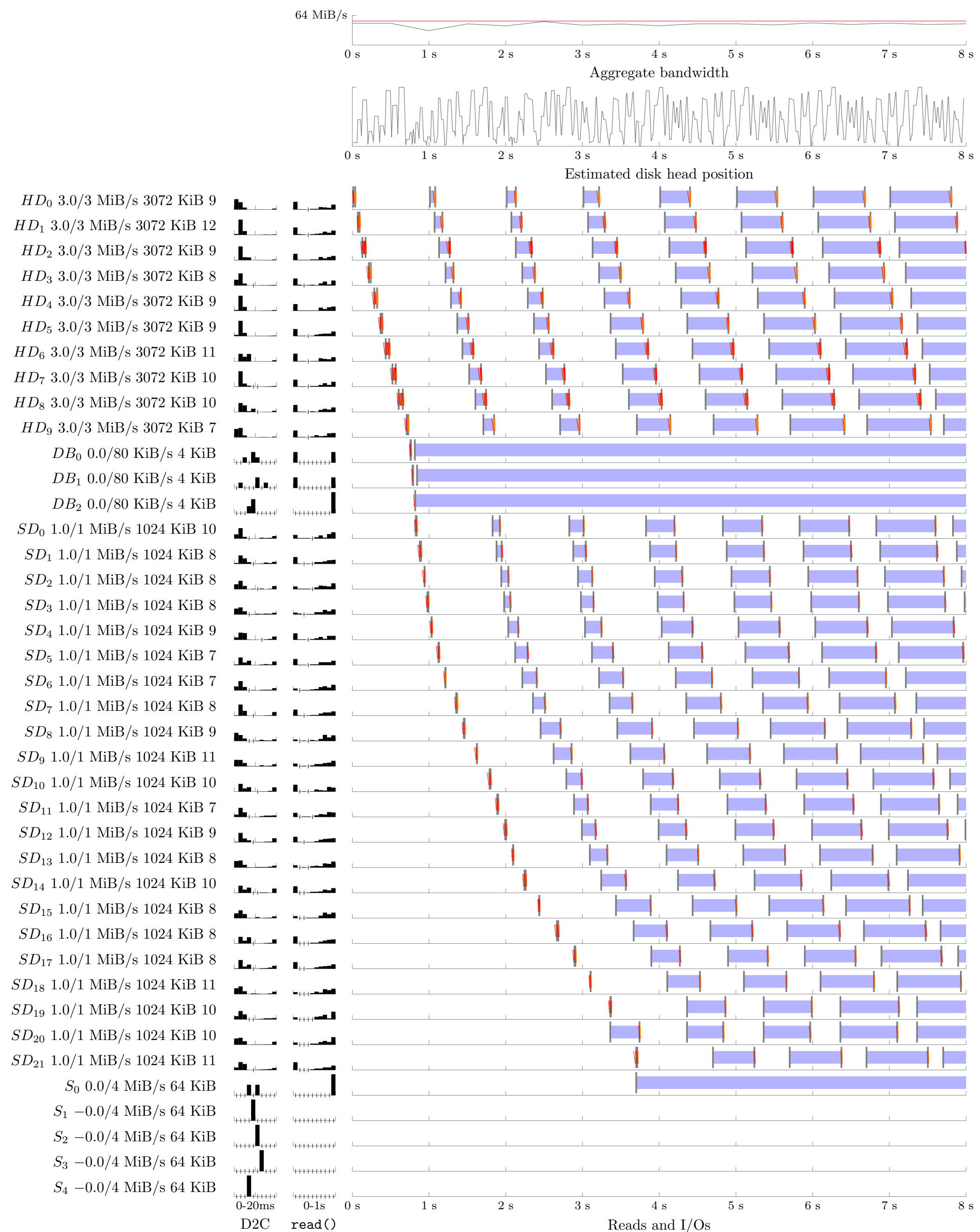


Figure 3.2: CFQ RT streams ends up with deadline misses

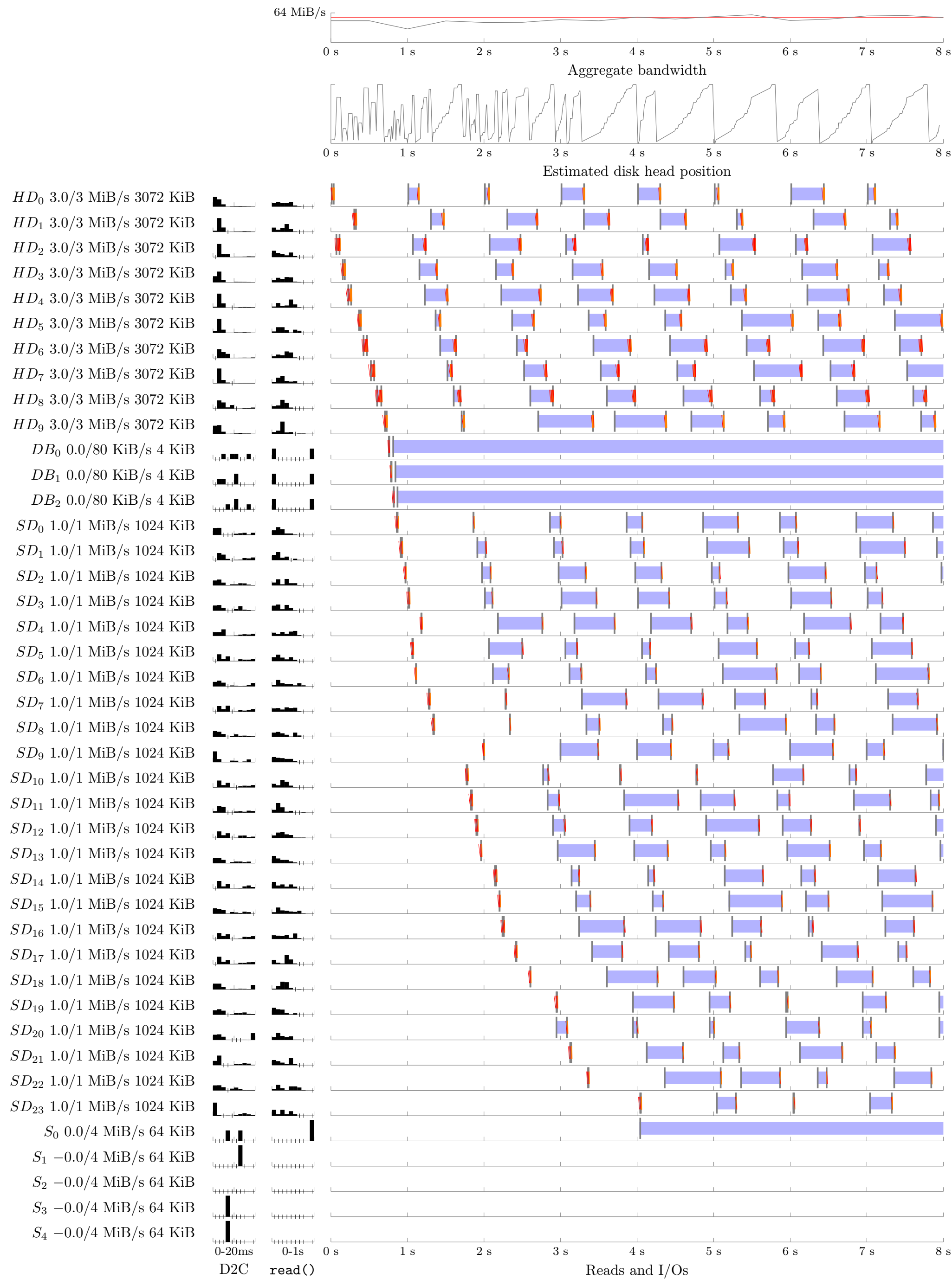
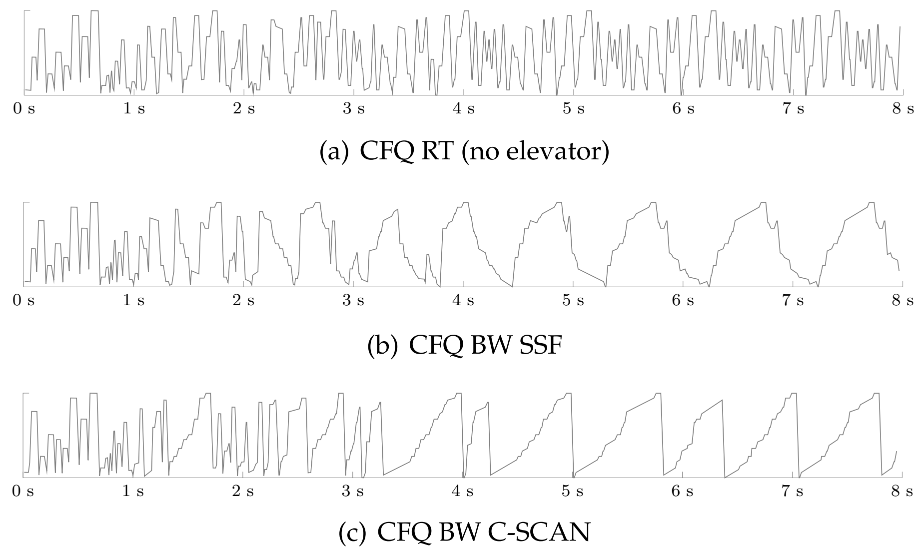


Figure 3.3: CFQ BW streams handled within deadline (C-SCAN)



(a) CFQ RT (no elevator)

(b) CFQ BW SSF

(c) CFQ BW C-SCAN

Figure 3.4: Disk head position with 34 media streams

3.3.2 Isolation

As put in the design requirements, there are two types of isolation we want from our system:

Best Effort Readers Should Not Affect Reserved Readers

If a BW schedule runs without any deadline misses in an isolated system where there are zero best effort readers, it should also do so in the presence of an arbitrary amount of best effort readers.

The first test to see if this goal has been reached, is a simple benchmark based on the previous section: we remove all best effort readers from the first scenario with deadline misses (35 reserved readers). If the number of deadline misses is consistently less, we know the goal has *not* been reached. The result in figure 3.5 shows little change in deadline misses when we remove all the best effort readers.

Another test for the same requirement is to have a scenario where best effort classes are scheduled more often, so we can verify that the preemption is working. We create a scenario where the base time slice of a BE class is one full second, i.e., the best effort readers would get exclusive access to the disk for one full second each. This would cause severe deadline misses when we set the scheduling deadline to 50 ms for two BW class readers at 3 MiB/s. A danger with preemption is that it may break the fairness property of the best effort class. We will therefore run this test with

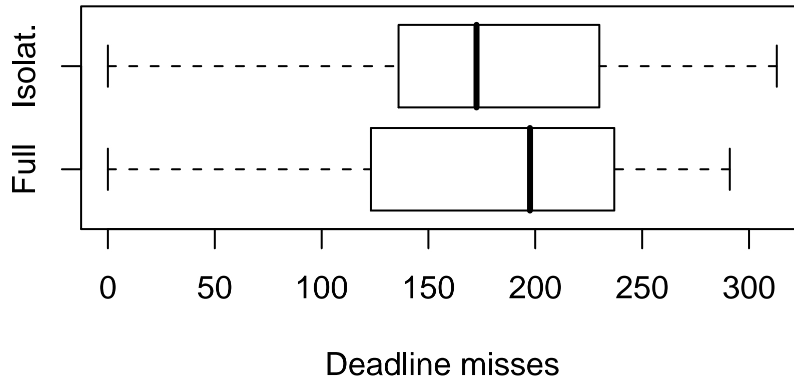


Figure 3.5: Boxplot of deadline misses for BW streams with ("Full") and without BE presence ("Isolated"). 60 runs of each scenario

and without BW readers to see if we find any hints that the *non-intrusive* design goal has been breached.

Stream	Latency			Bandwidth
	< 100 ms	100-1000 ms	1000 ms+	
Video BW3 A	100 %	0 %	0 %	3.0 MiB/s
Video BW3 B	100 %	0 %	0 %	3.0 MiB/s
Greedy BE4 A	96.43 %	0.42 %	3.15 %	48.73 MiB/s
Greedy BE4 B	96.91 %	0 %	3.09 %	49.65 MiB/s

Table 3.2: 3 MiB/s streams in BW class preempting long time best effort slices

The results in table 3.2 shows that preemption works; otherwise there would be latency of over 500 ms visible, as it is when we use the highest priority best effort class in table 3.3. There does not seem to be a major impact on the performance or fairness of the interrupted streams, but this may also be due to the fact that random unfairness will even out in the long run. As expected, the latency of the best effort streams was slightly worse when they were preempted.

Stream	Latency			Bandwidth
	< 100 ms	100-1000 ms	1000 ms+	
Video BE0 A	16.67 %	60 %	23.33 %	3.0 MiB/s
Video BE0 B	13.33 %	63.34 %	23.33 %	3.0 MiB/s
Greedy BE4 A	97.19 %	0.20 %	2.61 %	51.01 MiB/s
Greedy BE4 B	97.01 %	0 %	2.99 %	47.88 MiB/s

Table 3.3: 3 MiB/s streams suffers with highest priority best effort

Greedy Reserved Readers

A greedy BW class stream should not get more bandwidth than reserved if it would affect any best effort reader. To measure this, we have created a scenario with two greedy streams: one best effort and another reserved reader at 3 MiB/s. The initial bucket size for the reserved reader was 3 MiB.

The result was that the reserved reader got 3.08 MiB/s and the best effort class got 100.1 MiB/s. This means that the BW stream class got slightly higher bandwidth than it theoretically should, but within reasonable limits. As the main goal of limiting the bandwidth of the process is to prevent denial of service or starvation, as would happen if this was an RT class stream, we are satisfied with the result.

Other Potential Problems

It should be noted that these tests were run with the default 512 KiB maximum request size. The hardware on our system, which has LBA48 support, can handle up to 32 MiB large requests. The preemption does not preempt requests which are dispatched to the disk drive, consequently our process would not be able to dispatch the request before the request has completed by the disk. For very slow disk drives, such a large request will take some time. A system administrator must have this in mind if changing the block device queue tunable `max_sectors_kb`.

If a feature such as NCQ/TCQ is enabled, a similar issue arises: Several commands may be queued, and we have no impact on ordering of execution. The best we could do is to not insert a lower priority request while we are waiting for real time requests. During early testing of our system, we noticed that the firmware of our particular disks could delay requests by as much as 1 second. This is an issue for any QoS scheduler.

When a bandwidth-class queue preempts a best effort queue, we need two seeks; one to go to the stream, and then another one to go back to the best effort class later. Preemption might not always be necessary to

reach the deadline goals; and if that is the case, the preemption caused one more seek than required, which leads to worse performance. With the current design of the deadline list it is not possible to know whether we can postpone scheduling the reserved reader until the currently executing queue has expired. One of the reasons is that we do not have any estimate of how long each reserved reader can run each time.

3.3.3 Work-conservation

Over-provisioning

There are two work-conservation scenarios we must test when a reservation reader has requested more bandwidth than it uses (over-provisioning). In both scenarios, we must have at least one greedy reader to reach 100% utilization. We have used two greedy readers in order to check for fairness as well.

In the first scenario, the greedy readers are in the BE class with the default priority level (4), and in the second scenario they have a reserved bandwidth of 4 MiB/s and a scheduling deadline of 500 ms. To measure global throughput without our BW class, we have a baseline with only best effort readers as shown in table 3.4.

Stream	Reserved	Requested	Result	Latency
Video	BE4	8 MiB/s	8.0 MiB/s	313 ms max
Greedy A	BE4	∞	26.5 MiB/s	253 ms max
Greedy B	BE4	∞	26.5 MiB/s	243 ms max

Table 3.4: Baseline for over-provisioning evaluation

In both tests, the over-provisioning BW class reader has reserved 32 MiB/s, but it will only consume 8 MiB/s. It performs reads four times per second (2 MiB blocks) and has requested a 200 ms scheduling deadline.

Stream	Reserved	Requested	Result	Latency
Video	32 MiB/s	8 MiB/s	8.0 MiB/s	190 ms max (42 ms avg)
Greedy A	BE4	∞	25.5 MiB/s	779 ms max
Greedy B	BE4	∞	30.5 MiB/s	700 ms max

Table 3.5: Over-provisioning has little effect on global throughput with best effort readers

The results in table 3.5 and 3.6 both show aggregate bandwidth equal to or slightly above the baseline in table 3.4, i.e., utilization around 100%. The

Stream	Reserved	Requested	Result	Latency
Video	32 MiB/s	8 MiB/s	8.0 MiB/s	179 ms max (60 ms avg)
Greedy A	4 MiB/s	∞	4.1 MiB/s	787 ms max (487 ms avg)
Greedy B	4 MiB/s	∞	55.2 MiB/s	128 ms max (36 ms avg)

Table 3.6: Over-provisioning for BW class

conclusion is that work-conservation works even with over-provisioning, otherwise a drop in aggregate bandwidth should be visible. In this case, we reserved 24 MiB/s more than we used.

Fairness amongst the best effort readers in table 3.5 is slightly skewed. This may be an indication that the preemption algorithm does not give the preempted task sufficiently extra time on the next schedule.

When a process is allowed to be greedy beyond reservation, our design stops considering deadlines in the normal way. The deadline is set to the time when a claim for this bandwidth would be legitimate, i.e., when the number of tokens in the bucket is positive. This is why the deadlines for "Greedy A" in table 3.6 may seem broken, but because it has received more bandwidth than it requested, its buffers should cover the extra latency. We do, however, limit how many negative tokens we count, in order to limit the worst case scheduling delay in such a scenario.

Work-conservation with Non-Greedy Best Effort Readers

In the previous section, we studied work-conservation with over-provisioning. It is also important to consider another scenario, where we have best effort readers with limited bandwidth usage, and one or more greedy readers in the BW class.

We have created a scenario similar to the previous section, but limited the best effort readers to 8 MiB/s. The reserved reader still reserves 32 MiB/s, but will use anything it can get. The result should be that both best effort readers should get 8 MiB/s, because we know they *can* get that if the BW class was throttled at 32 MiB/s.

Stream	Reserved	Requested	Result	Latency
Greedy	32 MiB/s	∞	63.5 MiB/s	138 ms max (31 ms avg)
Video A	BE4	8 MiB/s	8.0 MiB/s	676 ms max (48 ms avg)
Video B	BE4	8 MiB/s	8.0 MiB/s	739 ms max (65 ms avg)

Table 3.7: Work-conservation for BW class

Table 3.7 shows that work-conservation works, the aggregate through-

put is well above the baseline. The other important goal here is that the work-conservation does not affect the performance of the best effort readers.

3.3.4 Request Deadlines / Variable Round Support

The last test is a schedule with a mix of 10 video streams and two best effort streams. We want to show how the scheduling deadline affects latency. In table 3.8, we have two streams with a scheduling deadline of 200 ms, and we see that data sometimes is returned more than 250 ms after the scheduling deadline for the two low-latency videos (A and B). The worst case was 283 ms (83 ms after the scheduling deadline). The user must consider the load in the BW class to estimate when the data will be returned. Our tests indicate that the data can be returned up to 300 ms after the scheduling deadline with a very heavy load, so as a rule of thumb, one could subtract 300 ms from the real deadline when setting a scheduling deadline.

The scheduling deadline is still an effective method for requesting lower latency service. In table 3.9, we lowered the scheduling delay to 50 ms, and the result is that we always get data within 250 ms. The performance of the other streams did not change much, therefore they are not included.

Stream	Bandwidth		Latency			
	Reserved	Result	Requested	< 250 ms	250-500 ms	500-1000 ms
Video A	3 MiB/s	3.0 MiB/s	200 ms	98.3 %	1.67 %	0 %
Video B	1 MiB/s	1.0 MiB/s	200 ms	91.6 %	8.33 %	0 %
Video C	1 MiB/s	1.0 MiB/s	800 ms	75.0 %	25.00 %	0 %
Video D	1 MiB/s	1.0 MiB/s	800 ms	98.3 %	1.67 %	0 %
Video E	1 MiB/s	1.0 MiB/s	800 ms	90.0 %	10.00 %	0 %
Video F	1 MiB/s	1.0 MiB/s	800 ms	26.7 %	73.33 %	0 %
Video G	3 MiB/s	3.0 MiB/s	800 ms	93.3 %	6.67 %	0 %
Video H	3 MiB/s	3.0 MiB/s	800 ms	96.7 %	3.33 %	0 %
Video I	3 MiB/s	3.0 MiB/s	800 ms	61.7 %	38.33 %	0 %
Video J	3 MiB/s	3.0 MiB/s	800 ms	46.7 %	53.33 %	0 %
Greedy A	∞	21.3 MiB/s	N/A	52.5 %	21.88 %	21 %
Greedy B	∞	24.2 MiB/s	N/A	58.6 %	20.44 %	19 %

Table 3.8: Latency vs. scheduling deadline

Stream	Bandwidth		Latency			
	Reserved	Result	Requested	< 50 ms	50-100 ms	100-125 ms
Video A	3 MiB/s	3.0 MiB/s	50 ms	45.00 %	46.67 %	8.33 %
Video B	1 MiB/s	1.0 MiB/s	50 ms	41.66 %	56.67 %	1.67 %
			...			

Table 3.9: All data returned within 75 ms after scheduling deadline. The rest of the system performed similar to table 3.8

3.4 Conclusion and Future Work

We have shown that the BW class implements all our design goals in a satisfactory way. The BW class gives higher throughput than CFQ in a streaming scenario, and it is able to meet deadlines and bandwidth requirements as long as reasonable limits are set.

Some of the benchmark highlights problems with the design, which had larger worst case effects than expected, others were known in advance:

Best effort deadlines The deadlines are best effort, and we do not detect multiple deadlines around the same time.

"Scheduling deadline" We interpret deadlines as the time for when a request should be *scheduled*, not when data should be *ready*. Both of these issues require the user to add some slack when setting the deadline.

Fairness in BW work-conservation Table 3.6 highlights a problem with work-conservation in the BW class: the extra bandwidth is not fairly divided between the greedy processes. As this was not a design goal, it should not be surprising, but it is worth noting that this happened by using C-SCAN, not just SSF.

Solid State Disk For Solid State Drives (SSDs) an EDF-based scheduler could be better—as seeks do not matter.

Chapter 4

Scheduling in User Space

In chapter 2, we identified an I/O scheduling problem which the I/O scheduler was unable to do anything about: because there was at most one request pending at any time, the scheduler could not do anything but dispatch the request directly. In figure 2.30, we showed that the time difference between a fully ordered read of all the data, and the order used today, was very large. We also studied how the file system works, so we know where data is stored, and we know that there are several reasons why this will cause seeking. In this chapter, we will put the pieces together and implement an alternative approach to this problem, through scheduling in user space. By scheduling in user space we mean that we will request data in the order which we believe causes least seeks. We will do the scheduling in user space because we believe this is the easiest way to achieve the best performance. Our goal is to improve the performance in the direction of the theoretical limit, while keeping the software simple and clean.

This chapter starts with a discussion of what kind of programs this technique can be applied to. Then, we will look into the information available for scheduling in user space, before we do a case study of the "tar" program. In the case study, we will design, implement and evaluate this technique. Finally, we will have a discussion around why we ended up with this solution, and the implications this has.

4.1 Applicability of User Space Scheduling

In order to increase performance by reordering requests, we obviously need an application that at times has multiple I/O operations to choose from. The potential improvement in execution time will generally depend

on how many I/O operations that can be reordered. This will often be a problem for applications depending on external input, such as interactive applications and databases; there is no way to know what data to read until a user requests it.

There is also a problem for applications with data dependencies; where one I/O operation decides which operation to do next, or where one write operation must be written to disk before another for consistency reasons. As an example of the former case, consider a binary search in a database index; the database only knows about one operation, *read the middle block*, while the next operation, *read the left branch* or *read the right branch*, depends on the data found by the first operation in the middle block. Other examples of dependencies are that we need to read a directory block to find a file's inode number, and to read the file inode before we can read the file data. In the utility we presented in section 2.3.11, we implemented basic functionality for data dependencies. A more advanced approach would include a full graph of dependencies, but using such a utility would likely be rather cumbersome because producing the graph is not a straight forward procedure.

For large files, we believe that there is usually not enough external fragmentation in modern Linux file systems to warrant reordering of requests *within the same file*. This will likely require intrusive changes in program flow, and yield relatively low performance gains. This means that we are limited to improving applications reading multiple files, and the smaller files, the more gain possible.

Another relevant question is whether it is likely that files are cached, i.e., if they have been recently used. A compiler such as `gcc`—or rather the preprocessor `cpp`—may seem like a good candidate, because it reads many small header files (around 20 for many standard C headers i GNU libc, almost 100 for GTK+). But in most workflows, header files will be cached and therefore the average gain over time will be low.

The last issue is whether the architecture of the program allows the change without refactoring the program. If the software needs to be rewritten completely, it is usually not worth it unless we expect a considerable gain in performance. We have used these guidelines to categorize a set of commonly used software into good and bad candidates for optimization, as shown in table 4.1.

Some of the programs, such as `ps`, hardly reads any files, except for libraries, which are most likely found in the page cache. The `vim`, `man` and `less` programs only read one file, so there are no files to reorder. The `cat` program *may* read multiple files, but they must be output in a specific order. This means that there are situations where improving `cat` is possible,

Good candidates	**Bad candidates**
`cp -r`	`Firefox`
`zip`	`vim`
`rsync`	`ps`
`tar`	`cat`
`scp -r`	`man`
`rm -r`	`locate`
`find`	`less`
`mv`	
`du`	
`Tracker`	
`File Manager`	
`ls -l`	

Table 4.1: Common Linux Software

but it would require buffering.

Most of good candidates for optimization through user space scheduling (see table 4.1) behave almost identically , i.e., they must read all data and metadata for a full directory tree, most of the job is known ahead, and it typically involves a large number of files. Most graphical file managers include these operations, and others, like thumbnail generation, which can be improved in the same way.

A few of the programs listed as good candidates, such as `ls -l` and `du`, only need to read metadata, which the ext4 file system has improved with inode readahead (section 2.3.10), but still some improvement should be possible. It is also important that such core utilities have a low overhead because they are often critical during recovery of overloaded systems.

`find` reads metadata just like `du`, but it is also often used to execute external software on a subset of files. If we believe this external program will read the files, it would be beneficial to order them so they are read in disk order.

4.2 Available Information

To do scheduling in user space, we need to know as much as possible about the *physical placement* of the data on the storage device. If the physical location is not available, we may at least be able to get the *relative placement*. We can then use this information to decide which operation to do next.

```
struct dirent {
        ino_t           d_ino;       /* inode number */
        off_t           d_off;       /* offset to the next dirent */
        unsigned short d_reclen;     /* length of this record */
        unsigned char  d_type;       /* type of file */
        char           d_name[256]; /* filename */
};
```

Listing 4.1: Linux `struct dirent`

4.2.1 Metadata Placement

Path traversal in Unix is done by using the `readdir` function (the system call is named `getdents`), which returns the `struct dirent` data structure. The `opendir` function, which uses the `open` system call, reads the directory inode, while `readdir` reads the data blocks of the directory file.

The attributes available in a `struct dirent` in Linux are shown in listing 4.1. As the listing shows, the kernel exposes information inode numbers in the `d_ino` attribute. In [28], it is proposed to get metadata for files in a directory by sorting the entries by inode number. This is proposed as a solution to the random order caused by hashing, and this reduces the number of seeks within a directory.

There is currently no method for mapping an inode number to a block offset, but this is not a major problem: In the file system implementations we consider here, ext4 and XFS, the ordering between inode blocks is the same as inode numbers. More formally, for any file a and b, we know that

$$\text{inode}(a) < \text{inode}(b) \Rightarrow \text{block}(\text{inode}(a)) \leq \text{block}(\text{inode}(b)) \tag{4.1}$$

This does not help us if we want to decide whether a data block operation should be scheduled before an inode block operation. As we remember from section 2.3.3 the inodes are clustered in groups, which are spread out all over the disk, with normal data blocks in between. This means that an *optimal* schedule would be to order all data in the same queue, and then read them with one full disk arm stroke. By using file system specific information such as the flexible block group size we *can* calculate the physical location, but we think this would usually not be a good idea. It is very file system specific and error prone, and we also believe the potential performance gain does not make it worthwhile.

Some file systems, such as ext4, also include the directory entry type (regular file, directory, link etc.) in the `d_type` attribute. If `d_type` has the

```
struct fiemap {
  u64 fm_start;  /* logical offset (inclusive) at
                  * which to start mapping (in) */
  u64 fm_length; /* logical length of mapping which
                  * userspace cares about (in) */
  u32 fm_flags;  /* FIEMAP_FLAG_* flags for request (in/out) */
  u32 fm_mapped_extents; /* number of extents that were
                          * mapped (out) */
  u32 fm_extent_count; /* size of fm_extents array (in) */
  u32 fm_reserved;
  struct fiemap_extent fm_extents[0]; /* array of mapped extents (out) */
};

struct fiemap_extent {
  u64 fe_logical;  /* logical offset in bytes for the start of
                    * the extent */
  u64 fe_physical; /* physical offset in bytes for the start
                    * of the extent */
  u64 fe_length;   /* length in bytes for the extent */
  u32 fe_flags;    /* FIEMAP_EXTENT_* flags for this extent */
  u32 fe_device;   /* device number for extent */
};
```

Listing 4.2: `FIEMAP` related data structures

value `DT_UNKNOWN` we must read the file inode to get this information. The `d_type` attribute is often useful if we are only doing path traversal. If we for example are looking for a named file within a directory tree, we only need to fetch the inodes for directories, and we can do this by looking for the type `DT_DIR`. On file systems without `d_type`, such as XFS, this process would require reading all inodes, including file inodes.

4.2.2 File Data Placement

A new system call was introduced with the ext4 file system, the `FIEMAP` [29] ioctl. It allows users to get the extent mapping of any file which the user can read. This means that we can avoid unnecessary long seeks by reading the file data in the same order as it is found on the disk. For older file systems, such as ext2/3 and XFS, there is similar functionality in `FIBMAP`, which returns the position of the requested logical block instead of a range of data, but this system call requires root privileges because of security problems [30]. `FIEMAP` is expected to be available for most file systems in a future version of the Linux kernel.

The `FIEMAP` data structures are shown in listing 4.2. An example of how

```c
/* Return the physical location of the first logical block
 * in a file.
 */
u64 firstblock(int fd) {
  u64 retval;
  struct fiemap *fiemap = malloc(sizeof(struct fiemap)
                                 + sizeof(struct fiemap_extent));
  if (!fiemap)
    err(EXIT_FAILURE, "Out␣of␣memory")

  fiemap->fm_start = 0; /* First logical block */
  fiemap->fm_length = 4096; /* We need to request at least one block */
  fiemap->fm_flags = 0;
  fiemap->fm_extent_count = 1; /* We have allocated 1 fiemap_extent */

  if (ioctl(fd, FIEMAP, fiemap) < 0 || fiemap->fm_mapped_extents == 0) {
    /* Here we should check errno and do a graceful recovery
     * for example consider using FIBMAP or inode number */
    retval = 0;
  } else {
    retval = fiemap->fm_extents[0].fe_physical;
  }

  free(fiemap);

  return retval;
}
```

Listing 4.3: Using `FIEMAP`

to use them with the `FIEMAP` ioctl is shown in listing 4.3, where we have implemented a function which returns the position of the first logical block of a file.

4.3 Implementation of User Space Scheduling

Our user space scheduling implementation uses a two-pass traversal over a directory tree, where the first pass fetches meta data and directories in C-SCAN order, and the second pass fetches file data in the order of the first data block of each file. Due to lack of information about directory data position, they are read in inode order during the first pass. If the Linux kernel API is extended to include this information, we could probably improve performance slightly more. Pseudo-code for the traversal algorithm is shown in listing 4.6. We will go into more detail in a practical example in section 4.5.

The user of the library implements the filtering, i.e., she decides which directories to enter and which files to read. During the second pass the user is called for each file to read, at the time which we consider it optimal to read that file. In listing 4.4 we have shown the complete implementation of the program `du`, which calculates the disk usage of a directory tree.

4.4 Future Proofing Lower Layer Assumptions

One problem with optimizing in the application layer is that we must make assumptions about the behaviour in lower layers, such as file system and disk. In this section, we discuss how to prevent this from becoming a problem in the future.

4.4.1 Placement Assumptions

The property in equation 4.1 may not hold for other file systems, especially those with *dynamic inode allocation*, like btrfs [31]. In [32], this feature is discussed for ext4, but it will break compatibility so it will likely be postponed until a new generation of the Linux extended file system is developed (i.e., ext5 or later).

For such file systems, we do not know the effect of inode sorting, it might be less efficient than using the inodes in the order in which they are received from the file system. If we fear this problem, one solution would

```
#include <iostream>
#include <libdir.h>

#define LINUX_STAT_BLOCK_SIZE 512

struct qdu : public libdir::traverser {
        size_t total_size;

        qdu() : total_size(0) {}

        bool add_directory(const libdir::dentry& de) {
                total_size += de.stat().st_blocks;

                return true; /* Enter directory */
        }

        /* Called when file metadata is ready */
        bool add_file(const libdir::dentry& de) {
                /* Sum sizes */
                total_size += de.stat().st_blocks;

                /* We don't want to read the file data */
                return false; /* Don't read file (add_file_data) */
        }

        /* Iff we return true from add_file,
         * this will be called when it is a good time for
         * reading file data.  Not used for du */
        void add_file_data(const libdir::dentry& de) { }
};

int main(int argc, char *argv[]) {
        qdu du;
        du.walk(argv[1]);

        std::cout << "Size: " << (du.total_size * LINUX_STAT_BLOCK_SIZE)
                / 1024 / 1024 << std::endl;
}
```

Listing 4.4: Using the traverser API

be to detect the file system type and disable sorting if it is an unknown file system, or a file system where sorting is known to be less efficient.

We can implement this check by using the `statfs` system call every time we find a new `st_dev` after calling `stat`. As we are unlikely to cross file system boundaries many times, this check will not have any noticeable overhead. One of the attributes of the `statfs` output is `f_type`, and we can use this value upon either a white list or a black list.

4.4.2 Rotational Disk

If the underlying block device is not a rotational disk, but a solid state drive, reordering does not help because there is no disk head to move. Doing reordering means that we spend unnecessary CPU cycles while the storage device may potentially be idling. We can avoid this by checking for a non-rotational device, which can be done by reading the file `/sys/block/<blockdev>/queue/rotational`. If it has the value "0", no sorting should be used.

4.5 Case Study: The `tar` "Tape" Archive Program

The `tar` program is archiving utility designed to store a copy of a directory tree with data files and metadata into an archive file. In this section, we will look at how we can create an optimized version of this program, based on the information we have written earlier in this chapter and chapter 2.

4.5.1 GNU Tar

`tar` is an old Unix utility that has been around for decades, so there are several implementations available. GNU Tar [33], which we tested, is the default implementation in GNU/Linux distributions. The pseudo-code for the traversal strategy in this implementation is shown in listing 4.5.

As we can see the utility does not use any metadata information to read the files, directories and metadata in a specific order, it always does a post-order traversal of the directory tree, and adds files in the order returned from the C functions seen in table 4.5.1.

Because GNU Tar was not developed as a two-part program, a tar library and a command line front end, we chose instead to use BSD Tar [34] as a reference implementation. BSD Tar includes the library `libarchive` that contains all functionality required for writing archives.

```
def archive(path):
    for file in path:
        stat file
        read file
        add to archive

    for subdirectory in path:
        stat subdirectory
        add to archive
        archive(subdirectory)
```

Listing 4.5: GNU Tar traversal algorithm

C function	Description	Reads/writes
`fstat`	Get metadata	inode
	(change times, mode, device and inode)	
`fopen`	Open file	inode
`fread`	Read from file	data, block map[a]
`fwrite`	Write to file	data, block map
`fclose`	Close file	
`opendir`	Open directory	inode
`readdir`	Get directory entry	data, (block map)
`closedir`	Close directory	
`ioctl FIBMAP`	Get block position	block map
`ioctl FIEMAP`	Get extent positions	block map

[a] By block map we mean direct blocks, indirect blocks and extent lists, depending on implementation. For small files/directories, we may find this information within the inode.

Table 4.2: Operations and data types

```
def archive(path):
    next = path
    do
        stat next
        if is_directory:
            add to archive(next)
            for file in next:
                inode_cscan_queue.add(next)
        else:
            FIEMAP next
            block_sort_queue.add(next)
    while (next = inode_cscan_queue.next())

    flush()

def flush():
    for file in block_sort_queue:
        add to archive
```

Listing 4.6: Our new tar traversal algorithm

4.5.2 Design and Implementation

We do not aim to implement a complete version with all the functionality of tar, just the standard function of creating a new archive of all the contents in a directory tree. To get access to standard data structures such as trees, we chose C++ as the implementation language. The algorithm of our implementation is shown in listing 4.6.

We have chosen to first traverse the directory tree, and then read all the files. The directory tree is traversed in inode order, but we do not consider the location of directory data blocks. This because it is currently not possible to use `FIEMAP` on directories, just files. It also keeps the implementation simple, and we believe that the relative gain is low (there are generally more files than directories). Because new inodes are discovered during traversal, we cannot pre-sort the list. Therefore we use a C-SCAN elevator that stores all unchecked paths sorted by inode number. The inode number is a part of the information from the `readdir` function.

The file list is a sorted list, ordered by block number. The block number is retrieved by using FIEMAP on the file, as shown in listing 4.3. Because the block queue does not change while traversing it, a simple sorted list is used.

If an upper bound of memory is wanted, the `flush()` method may be called at any time, e.g., for every 1000 files or when N MiB RAM has been

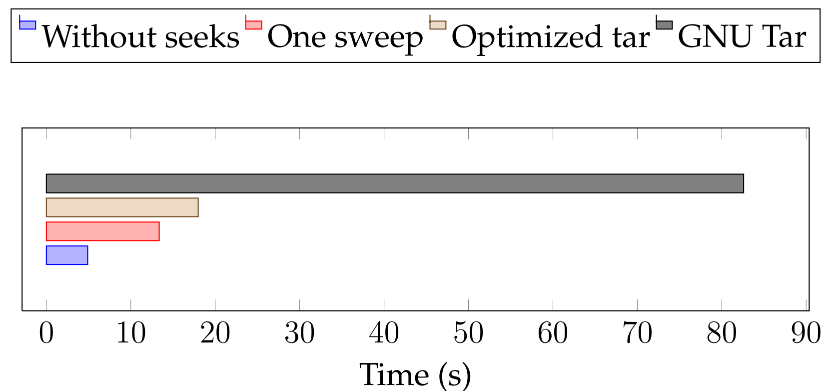


Figure 4.1: Speed improvement in context

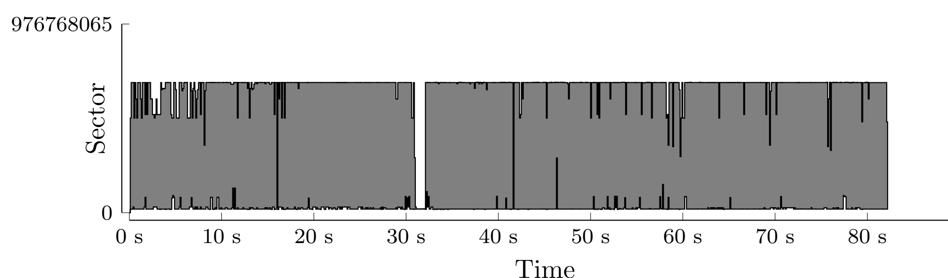
used. We store a couple of hundred bytes for each file, such as file name, parent directory, inode and block number. In our tests we did not use any limit on memory, so around 6 MiB of memory was used for the largest directory tree (22 500 directory entries).

4.5.3 Testing and Evaluation

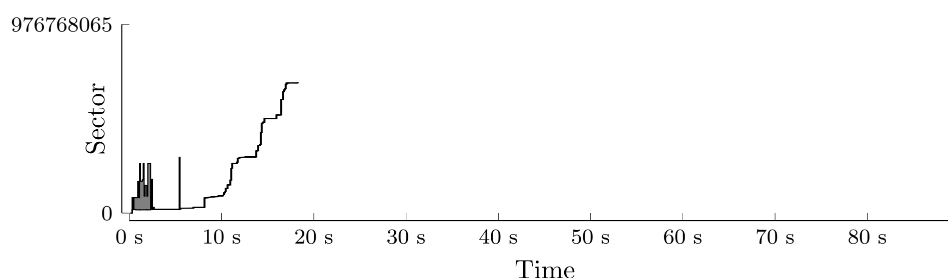
By running our new version on a large directory tree (22 500 files, the Linux kernel source) on the ext4 file system, we get an average runtime of 17.98 seconds for five runs. This is fairly close to the best case schedule (as defined in section 2.3.11): We have removed 78.2% of the original runtime of 82.6 seconds, while the upper limit is 83.8%. In figure 4.1, we have put the numbers in context. We used the same disk in all tests in this thesis (see table 2.1). Remember that "One sweep" is somewhat faster than theoretically possible, because we have several data dependencies (directory data blocks must be read to find new files and subdirectories, inodes must be read to find file data). The "As one file" number is not possible to reach by scheduling in user space, since it requires the file system allocator to place the files and metadata very tight, which it has not done. Because we have only tested our software on *one* directory, further evaluation and verification is required.

4.5.4 Verification by Visualizing Seeks

In figure 4.2, we illustrate the runtime and seek footprint of the two different implementations, by showing the maximum and minimum disk sector accessed for 100 millisecond intervals. The data for the figure was



(a) GNU Tar



(b) Improved tar

Figure 4.2: Upper and lower bound of disk head every 100 ms

recorded using `blktrace`. We could not include the full seek pattern, because there were too many seeks for the GNU Tar implementation. Therefore, we ended up showing only the upper and lower bound of the disk head, even though the finer details are lost.

As we can see, the footprint of the improved solution is radically different from the original solution. Almost all seeks are going in the same direction, except for some directory blocks which need to be read before new inodes are discovered. Our implementation orders directories by inode number, which may be one of the reasons for the seeking done during the directory traversal the first few seconds.

After the directory tree has been traversed, there does not seem to be any unnecessary seeks at all. This means that the file system allocator did not have to resort to any file fragmentation in order to fit the file on the file system. Both XFS and ext4 have implemented delayed allocation, so the full size of the file is known before the physical blocks of the file are allocated.

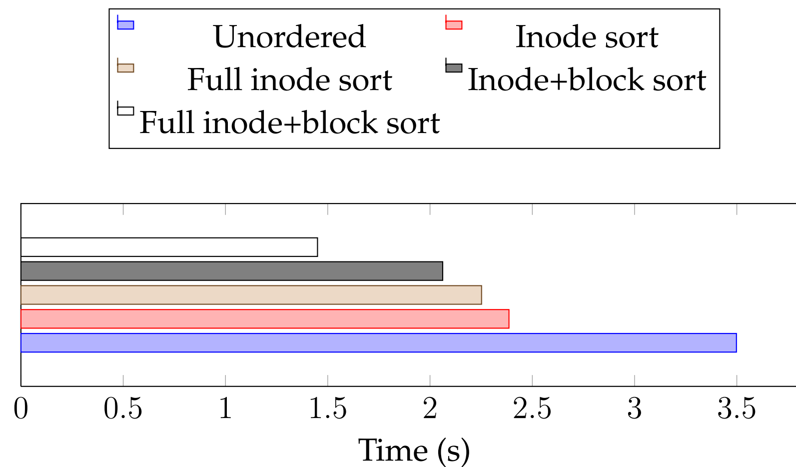


Figure 4.3: Sorting alternatives compared (reading 1000 files)

4.5.5 Partial Sorting

The implementation we have tested so far read the full directory tree and ensured that both metadata (inodes) and file data were read in the correct order. In figure 4.3, we have shown the performance of various alternative implementations of `tar` with only partial sorting, i.e., sorting only by inode number, or sorting only one directory at a time. These tests were ran on a directory tree with 1000 files. The alternative algorithms we tested were:

Unordered This implementation read all the metadata for all files in a directory in the order which they were retrieved from the file system. Then it read the file data in the same order, before recursively repeating the procedure for each subdirectory.

Inode sort This implementation sorted the directory entries by inode before reading the metadata. It also read the file data in the same order, i.e., sorted by inode.

Full inode sort This implementation traversed the full directory tree before reading all the metadata ordered by inode, and then it read all the file data ordered by inode.

Inode+block sort This implementation read all the metadata for a directory ordered by inode. Then it read all the file data for the file in the directory, ordered by the position of the first block of each file. This procedure was repeated for for each subdirectory.

Full inode+block sort This implementation traversed the full directory tree and read all the metadata ordered by inode. Then it read all the file data ordered by the first block of each file.

The numbers clearly shows us that doing all steps of the procedure gives the best performance, but also that even the simplest change, ordering directory entries by inode, helps a lot. This means that there is usually a high correlation between the inode number and the position of the file data.

4.5.6 Aging vs. Improvement

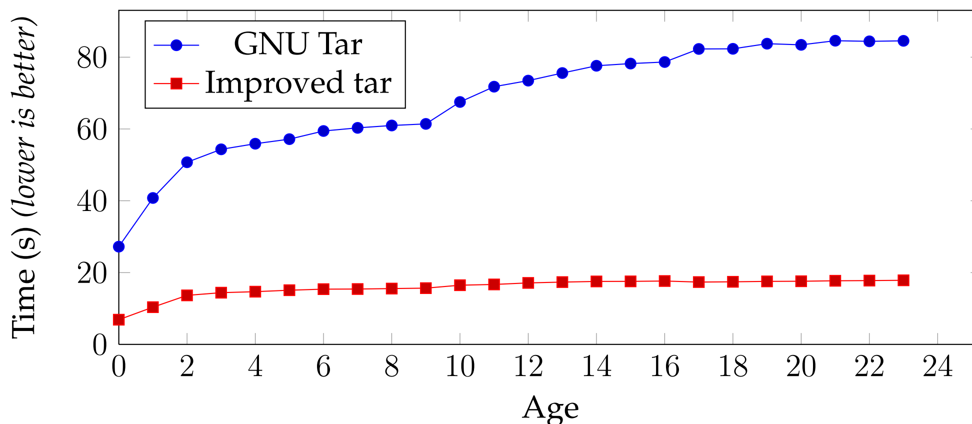


Figure 4.4: Performance as file system ages

An interesting subject is how the performance improvement changes as a file system ages. To test this, we ran our improved tar and GNU Tar 5 times throughout the aging procedure, which is described in appendix A.1. In short, our aging procedure replays development activity on the kernel source tree by incrementally checking out new versions. The archiving processes were run on a full Linux kernel source, i.e., about 22 500 files, and we used the ext4 file system. In figure 4.4 we have shown the average runtime for the 5 runs.

The amount of change in each in step in the aging procedure is different, which is why step 1, 2, 10 and 17 are especially steep for GNU Tar. What is interesting to see in this graph, is that the age has much less impact on the improved implementation. As the file system ages, the improvement factor increases from 3.95 to 4.75.

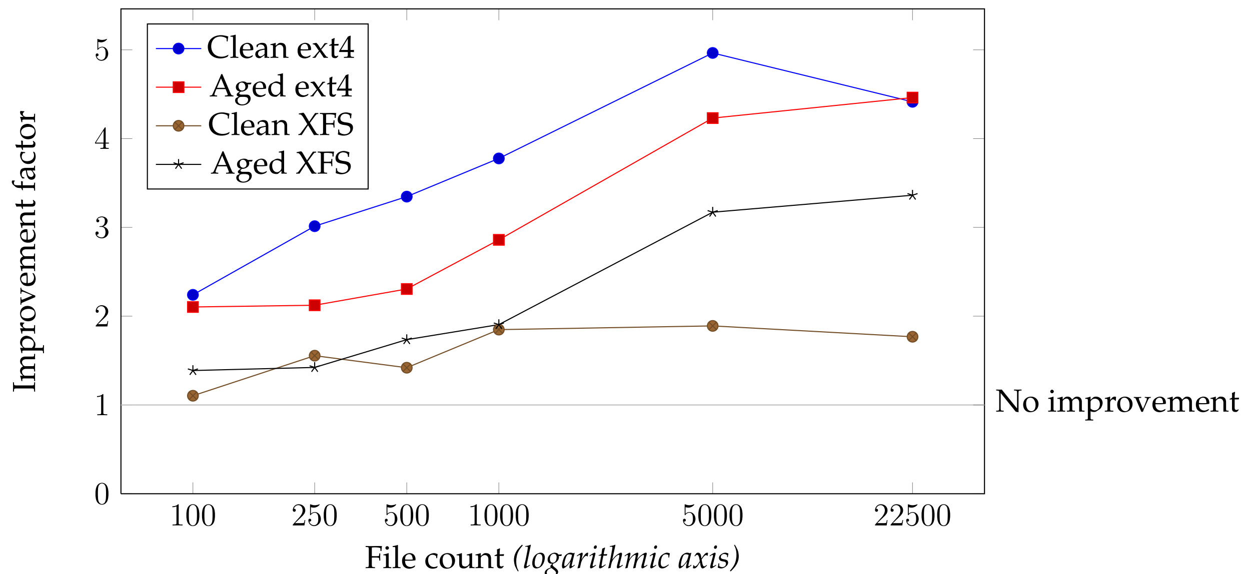


Figure 4.5: XFS vs. ext4, vs. directory size vs. age

File count	100	250	500	1000	5000	22500
Average size	8 KiB	24 KiB	10 KiB	10 KiB	21 KiB	18 KiB

Table 4.3: Average file size for the different data sets

4.5.7 File Systems, Directory Sizes and Aging

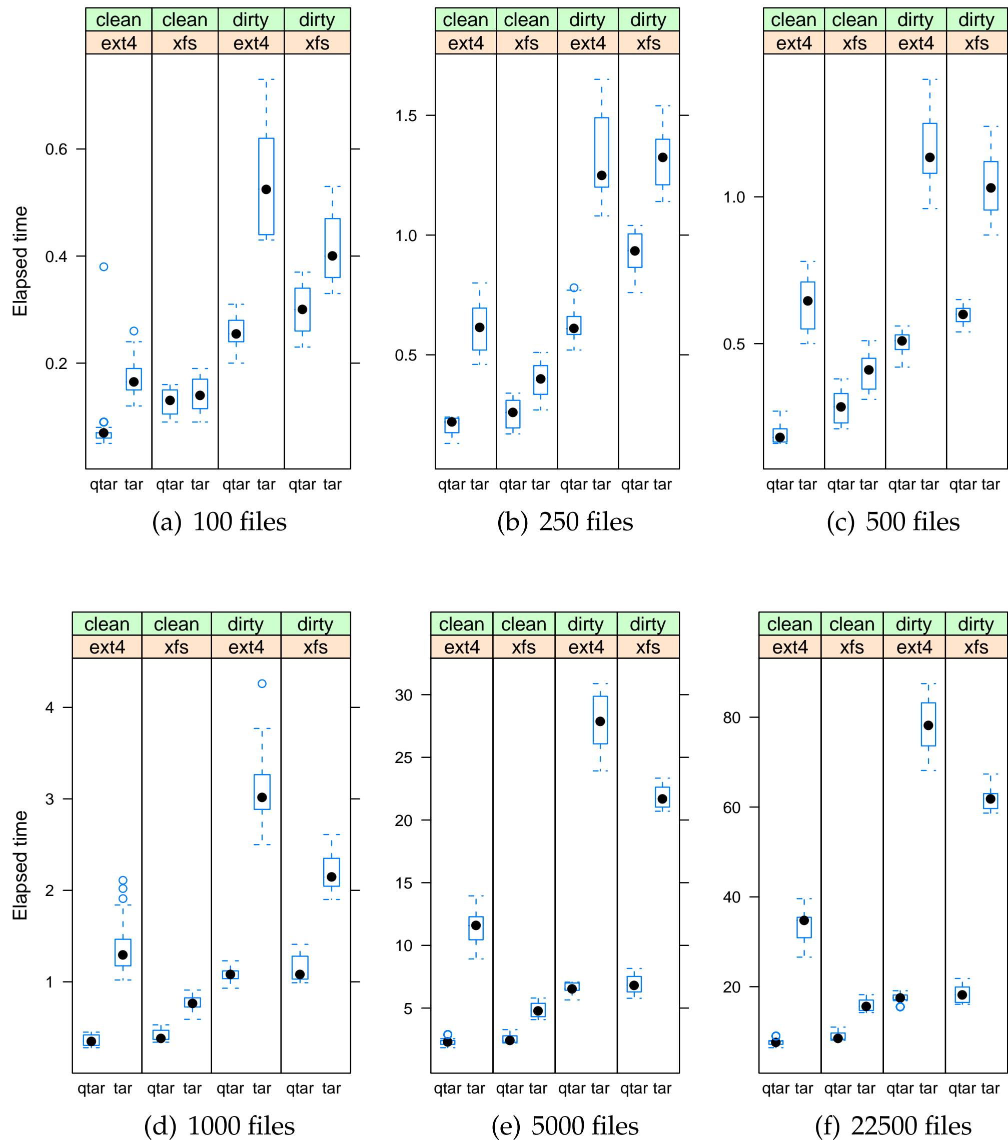
In figure 4.5, we show the results of comparing the runtime from GNU Tar with our improved implementation, e.g., mean time of GNU Tar divided by the mean time of our implementation. A value above 1 means that our implementation was faster.

The tests were run 5 times, and average runtime was used. Each test was run on 16 different directories (which were created in the same way, but at different times). We also ran the tests on 16 aged directories, i.e., each data point is the average of 80 runs. We did not run the tests on smaller directories because the runtime was already very low (between 0.05 and 0.5 seconds). In total, each implementation was tested on 384 different directories.

The goal of this test was to evaluate the impact of directory size, file system and aging. The tests reveals that we can expect more improvement on the ext4 file system than on XFS. Directories on XFS which have had little activity after creation ("clean" directories) are difficult to improve, for the smallest directory (100 files), we only got a 10% improvement on

the average values. The same data can be seen in figure 4.6, but there with absolute timings for both implementations. In this plot, we have named our implementation "qtar", and GNU Tar is written as "tar". Note that the variance for the boxplot mainly shows variance in the file system allocator, not variance in runtime with the same directory.

An important observation from this test is that the average runtime for the improved implementation was better for all directories we tested, by at least 10%.



(a) 100 files

(b) 250 files

(c) 500 files

(d) 1000 files

(e) 5000 files

(f) 22500 files

Figure 4.6: Absolute timings (each boxplot represents two implementations on 16 directories with 5 runs)

4.6 Discussion: Alternatives

The general opinion is that I/O scheduling belongs in the I/O scheduler, so we need to explain why this is not a feasible solution for optimal performance, at least with the APIs available in Linux today.

The main argument against doing scheduling in user space is that it adds complexity. This is true, like all optimization techniques, user space scheduling is a two sided sword, where you get better performance but more complex logic. This technique may be compared to using specialized multimedia operations in the video decoder. Therefore, we believe that for some software, user space scheduling can be justified:

- This is the core functionality of the software (e.g., backup software)
- The software has many users, so the total time saved would overweight the additional software complexity
- Performance in the software is critical

Another argument is that the properties of the underlying storage device may be different, i.e., this technique does not apply to solid state disks. It is likely that sorting would only slow down the application. We have already discussed and proposed a solution to this in section 4.4.

A solution where the kernel does the sorting could be implemented in several ways. An application can ensure that the I/O scheduler has multiple files to choose between by using the existing asynchronous APIs. There are several problems with this solution, which leads us to believe that such a solution would be slower and more complex:

Complexity The application must now handle multiple buffers and multiple pending asynchronous requests, which is more complex than sorting a list of files.

Performance limited by memory The performance depends on how deep the I/O queue is, i.e., how many operations there are to choose from. For file data, we must allocate buffers for all the files we want to have pending. The buffers must be as large as the complete file, if we use smaller buffers we may end up with more seeks, because requests for one file are interleaved with the requests from other files.

If we want a queue depth of 500, and the average file size is 20 KiB, we would need 4 MiB RAM. If we use 200 bytes ram per file for the user space solution, we could get the equivalent of a queue depth of around 20 000 with the same memory limitation.

Performance limited by scheduler The I/O scheduler assigns deadlines to these operations. The deadlines are there to prevent starvation, but in this case we want to delay requests as long as possible if we can prevent seeks.

Performance for metadata Another performance issue is the lack of asynchronous APIs for metadata operations. There are several projects for implementing such APIs, such as fibrils [35].

We have tested one of these, but ran into another issue: Locking inside the kernel effectively serializes operations such as `open` and `stat` on files in the same directory, the operations hold the lock `i_mutex` on the parent directory of the file. This means that the I/O requests will arrive one at a time, so the scheduler will not be in a better position to make decisions than before.

An alternative, which does not require modifying applications, is to alter the order in which files are returned from the kernel in the `readdir` operation [28]. This helps to some extent, but there are issues with this solution too:

No full tree sorting We will be limited to sorting the files for one directory at a time, while our proposed implementation sorts all files we are going to read. As shown in figure 4.3, not traversing the full directory tree decreases the performance significantly.

Decrease in performance of unrelated software The sorting would be done for all software, including software which is only interested in file names. This would add a slight overhead for such programs. There would be unnecessary sorting done, which requires CPU time, and because we need to support seeking within a directory, a complete copy of the directory must be kept in RAM.

Sorting on inode only We are limited to sorting on inode only, because we need to read all metadata to sort by block number. As shown in figure 4.3, just sorting by inode number is much slower than also sorting by block number.

One might argue that readdir could fetch metadata for all files, but this will add an unacceptable penalty for software which is only interested in the file names and not in reading the files.

A special purpose kernel API could be implemented for this operation, but it is not a viable solution, because we would need to move user space

logic into the kernel. For example: A file manager which generates previews of all files in a directory would only be interested in the embedded thumbnails for JPEG images (a small part of the file), and the first page of a multi-page document.

4.7 Summary

We have discussed how to identify programs which can be optimized by user space scheduling, and what information is available for doing scheduling in user space. We then showed how to avoid a potential negative performance impact in case of any unanticipated future developments, such as new file systems. In listing 4.4, we showed that it is possible to create an abstraction which hides the complexity of use space scheduling.

In section 4.5, we did a case study where we designed, implemented and evaluated the technique by implementing an archive program. We evaluated the technique in a wide range of scenarios, and the numbers always showed improvements, especially for larger directories. Finally, we discussed alternative approaches and why we ended up doing the scheduling in user space.

Chapter 5

Conclusion and Future Work

In this thesis we have studied the major components that affect disk I/O scheduling, and suggested improvements on multiple layers. We have shown that there is potential for improvements both in the Linux I/O scheduler, but also that there are times when the low level I/O scheduler is not enough. We have therefore introduced user space scheduling, and shown its potential.

I/O schedule visualization In chapter 2, we designed and implemented a new way of illustrating I/O schedules, giving a good overview when studying or debugging schedulers, file systems, applications and interaction between the layers.

This created the foundation for the next two chapters which allowed us to improve both the I/O scheduler and application layer. The utility was helpful both for locating problems with the existing solutions and during development of the software in this thesis.

QoS and performance improvements for CFQ In chapter 3 we implemented a practical approach for improving performance and QoS support in a scenario where the CFQ scheduler is used in an environment with bandwidth or deadline-sensitive applications such as multimedia.

Our results have shown that we get better performance by using a C-SCAN elevator between the streams. We have also shown that our new class gives the user control over latency and bandwidth (QoS). This class provides the user an efficient control of not only bandwidth, but also latency.

User space scheduling In the previous chapter we claimed that the options for the file system allocator and I/O scheduler are limited, and

that applications that read multiple smaller files and want maximum performance can achieve this by doing scheduling in user space. We have created a utility for evaluating the potential gain, written a guide for implementing the technique and shown a practical example with a considerable improvement.

The results show that there is a very large potential gain in performance compared to naive directory tree traversal. This should be a valuable contribution to specialized software such as backup programs.

While writing this thesis, we got several ideas for future work and ideas for new directions. A few of them are problems with the existing implementation, others are simply related ideas but that were considered out of scope for this thesis.

- In the BW class, two measures should be taken in order to decrease deadline misses:
  - Estimate the time it would take to service each process
  - Calculate the deadline for scheduling based on this information and other pending processes
- A desktop client version of the graphs shown in chapter 2, preferably in real time with little overhead.
- Fairness in the work conservation mode of the CFQ BW class should be implemented.
- A more extensible API for setting I/O priority/QoS parameters is needed, perhaps by integration with cgroups [36].
- If a solid state disk device is detected, the BW class should not use the sorted tree, just the deadline tree, and the anticipation should be disabled.
- We would like to see some of the core utilities in operating system utilizing the technique from chapter 4, and also graphical file managers.
- We did not look at improving the file system allocator and write patterns, but there are several ideas that would be interesting to investigate. In some cases file access patterns are known in advance, and

repeated many times. If the file system allowed manual, or at least hinted, file and inode placement, one could potentially optimize for the normal access pattern. Some examples include:

- All the files read when starting the software “Firefox” could be allocated back-to-back, allowing faster startup. This could either be done by the program package installer, or by observing application access patterns.
- Another example would be to place specific files near the outer edge of the disk for faster throughput, or closer to the center for lower latency.
- Inode tables seems like they are optimized for the rather special case of hardlinks, so placing inodes inside directory tree files could be an alternative.
- Meta data and file data for active files, which are written to often, such as log files, could be placed close to the file system journal.

In conclusion, we have shown that I/O performance is not something that only the I/O scheduler can improve upon, but that all the layers up to the application layer make up a big impact on the disk I/O performance.

Appendix A

Creating a Controlled Test Environment

Because file system behaviour often changes as it ages, we need to run our tests on an aged file system. Such a system could be created using traces, or by using a script. We have chosen to use a script because this gives us complete control over what kind of files we have and how they are created.

A.1 The Aging Script

Our script creates four types of work loads:

1. Large files created quickly with no concurrent activity
2. Directory trees created quickly and then never touched again
3. Directory trees with large amounts of file creations and deletions interleaved with activity on other files over time
4. Large files grown over time

The directory trees are created by checking repeatedly checking out newer versions of the Linux source code, from version 2.6.17 to 2.6.18-rc1, via all release candidates up to 2.6.18, and so on until 2.6.20. In total 24 different versions are checkout out. This could be considered as replaying a trace. In total this aging procedure creates 60 885 and deletes 39 510 files for each Linux source directory.

We also create clean checkouts of the Linux kernel source, without any aging. This means the file system has very good opportunity to locate

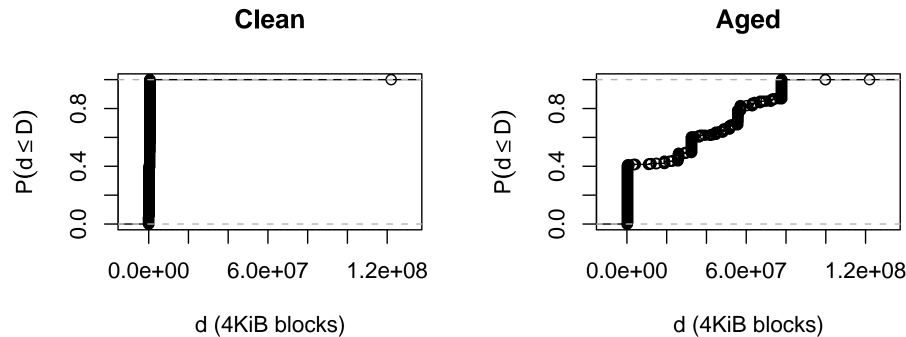


Figure A.1: Distance between inode and corresponding file

related files close to each other, so this directory may be considered the "worst case" for our testing. By using only parts of this directory tree, e.g. `linux-2.6/fs`, we can control the number of files and subdirectories.

For the media streams we create the file with no concurrent activity, 40 files of 8 GiB each were created. We also create some large files over time to fill the file system up to 93%, to make sure some of the inner tracks are accessed too,

During this aging procedure our file system utilizes up to 80% of the disk capacity. We keep the utilization below this limit because we are not interested in measuring the performance of the block allocator under pressure, we just want a realistic aged file system for our benchmarks.

The resulting file system allows us to test different kinds of files created in realistic environment, and shows us how the different allocation strategies influence our algorithms.

A.2 Aging Effects

To see how effective the aging procedure have been, and how different a clean directory is to an aged, we measured the distance between different in both data sets. A few selected observations are shown as ECDF plots.

Figure A.1 shows that the file data for a file in the clean directory is always within 2 GiB of the block containing the inode, which is pretty close. The dirty directory, created at a later point in time, shows that inodes are often allocated far away from the data: 60% of the inodes are 76 GiB or farther away from the corresponding file data.

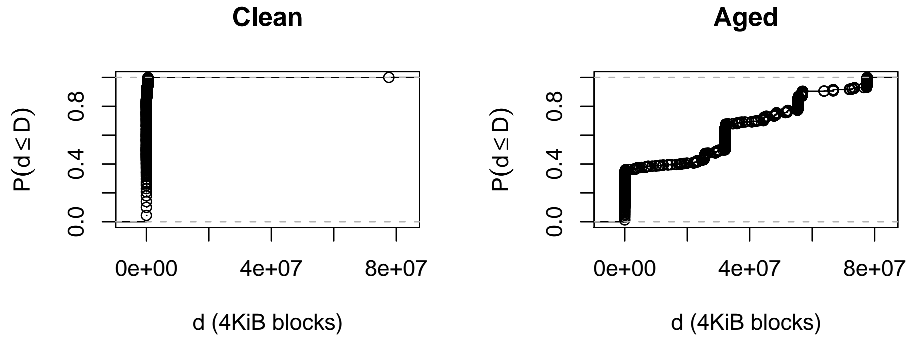


Figure A.2: Distance between data blocks of sibling files

If we fetch all metadata (inodes) during directory traversal it is interesting to see how far away the inodes are from the directory data blocks, which is the data we need to read to find the inode number. We observe that for both the clean and aged directory 99% of the blocks are very close. This is most likely due to the fact that few directories are created during the aging process, the directory structure in the Linux kernel history is almost static.

Figure A.2 shows that for the clean directory, 100% of the files in each directory a median distance for the first data block within 2 GiB. With the dirty directory we have the same issue as before, almost 60% of the siblings has a median distance of over 76 GiB.

In conclusion, we can see that aging has a big impact on data placement. This makes it important to do evaluation on both new and aged directories.

Appendix B

Source Code

Kernel patches and source code are available at the following address:
`http://www.ping.uio.no/~chlunde/master/`

Appendix C

List of acronyms

AS Anticipatory I/O Scheduler

BE Best Effort

BW Bandwidth

C-SCAN Circular SCAN

CFQ Complete Fairness Queueing

CPU Central Processing Unit

EDF Earliest Deadline First

FCFS First-Come, First-Served

FIFO First-In, First-Out

GDT Group Descriptor Table

IPC Inter-Process Communication

LBA48 48 bit Logical Block Addressing

LRU Least Recently Used

NCQ Native Command Queuing

NOOP No Operation

QoS Quality of Service

D2C Dispatch-to-Completion

RT Real-Time

SSD Solid State Drive

SSF Shortest Seek First

TCQ Tagged Command Queuing

VBR Variable Bit-Rate

VFS Virtual File System

WAL Write Ahead Logging

dcache dentry cache

Bibliography

[1] Professor David Patterson. Latency lags bandwidth. In *ICCD '05: Proceedings of the 2005 International Conference on Computer Design*, pages 3–6, Washington, DC, USA, 2005. IEEE Computer Society. doi:http://dx.doi.org/10.1109/ICCD.2005.67.

[2] Moonish Badaloo. An examination of server consolidation: trends that can drive efficiencies and help businesses gain a competitive edge. *White paper on IBM Global Services*, August 2008. Available from: http://www.ibm.com/systems/resources/systems_virtualization_pdf_GSW00579-USEN-00.pdf [cited May 2009].

[3] Seagate. Cheetah 15k.6 data sheet. Available from: http://www.seagate.com/docs/pdf/datasheet/disc/ds_cheetah_15k_6.pdf [cited May 2009].

[4] Seagate. Barracuda 7200.11 serial ata product manual. Available from: http://www.seagate.com/staticfiles/support/disc/manuals/desktop/Barracuda7200.11/100452348g.pdf [cited May 2009].

[5] Andrew S. Tanenbaum. *Modern Operating Systems*, pages 319–322. Prentice Hall International, second, international edition, 2001.

[6] Robert Love. *Linux Kernel Development*, pages 245–249. Novell Press, second edition, 2005.

[7] Jonathan Corbet. Trees ii: red-black trees. *LWN.net*, June 2006. Available from: http://lwn.net/Articles/184495/ [cited May 2009].

[8] Leo J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. pages 8–21, Oct. 1978. doi:10.1109/SFCS.1978.3.

[9] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*, pages 87–88. O'Reilly, 3rd edition edition, 2006.

[10] Sitaram Iyer and Peter Druschel. Anticipatory scheduling: a disk scheduling framework to overcome deceptive idleness in synchronous i/o. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 117–130, New York, NY, USA, 2001. ACM. doi:http://doi.acm.org/10.1145/502034.502046.

[11] Nick Piggin. Anticipatory io scheduler. *Linux Kernel Source v2.6.29*, 2003. Available from: Documentation/block/as-iosched.txt.

[12] Jens Axboe. *blktrace manual*, 2008.

[13] Alan D. Brunelle. Block i/o layer tracing: blktrace. April 2006. Available from: http://www.gelato.org/pdf/apr2006/gelato_ICE06apr_blktrace_brunelle_hp.pdf [cited May 2009].

[14] Jens Axboe. *blkparse manual*, 2008.

[15] Paolo Valente and Fabio Checconi. High throughput disk scheduling with deterministic guarantees on bandwidth distribution. Available from: http://algo.ing.unimo.it/people/paolo/disk_sched/bfq-techreport.pdf [cited May 2009].

[16] Vivek Goyal. Available from: https://lists.linux-foundation.org/pipermail/containers/2009-January/015402.html [cited May 2009].

[17] Andrew S. Tanenbaum. *Modern Operating Systems*, pages 732–747. Prentice Hall International, second, international edition, 2001.

[18] Gadi Oxman. The extended-2 filesystem overview. August 1995. Available from: http://www.osdever.net/documents/Ext2fs-overview-0.1.pdf [cited May 2009].

[19] Daniel Phillips. A directory index for ext2. In *ALS '01: Proceedings of the 5th annual Linux Showcase & Conference*, pages 20–20, Berkeley, CA, USA, 2001. USENIX Association.

[20] Robert Love. *Linux Kernel Development*, pages 223–224. Novell Press, second edition, 2005.

[21] Aneesh Kumar K.V, Mingming Cao, Jose R Santos, and Andreas Dilger. Ext4 block and inode allocator improvements. In *Proceedings of the Linux Symposium*, pages 263–274, Ottawa, Ontario, Canada, 2008.

[22] Andrew S. Tanenbaum. *Modern Operating Systems*, pages 721–722. Prentice Hall International, second, international edition, 2001.

[23] Stephen C. Tweedie. Journaling the linux ext2fs filesystem. In *The Fourth Annual Linux Expo*, Durham, North Carolina, May 1998. Available from: `http://e2fsprogs.sourceforge.net/journal-design.pdf` [cited May 2009].

[24] Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Analysis and evolution of journaling file systems. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 8–8, Berkeley, CA, USA, 2005. USENIX Association.

[25] Jonathan Corbet. On-demand readahead. *LWN.net*, May 2007. Available from: `http://lwn.net/Articles/235164/` [cited May 2009].

[26] Ketil Lund and Vera Goebel. Adaptive disk scheduling in a multimedia dbms. In *MULTIMEDIA '03: Proceedings of the eleventh ACM international conference on Multimedia*, pages 65–74, New York, NY, USA, 2003. ACM. `doi:http://doi.acm.org/10.1145/957013.957024`.

[27] Carl Henrik Lunde. Re: [patch 2/3] i/o bandwidth controller infrastructure. *Linux Kernel Mailing List*, June 2008. Available from: `http://lkml.org/lkml/2008/6/18/260` [cited May 2009].

[28] Theodore Ts'o. Re: [bug 417] new: htree much slower than regular ext3. *Linux Kernel Mailing List*, March 2003. Available from: `http://lkml.org/lkml/2003/3/7/348` [cited May 2009].

[29] Mark Fasheh. Fiemap, an extent mapping ioctl, 2008. Available from: `http://lwn.net/Articles/297696/` [cited May 2009].

[30] Alan Cox. Why is fibmap ioctl root only? *Linux Kernel Mailing List*, 2007. Available from: `http://lkml.org/lkml/2007/11/22/97` [cited May 2009].

[31] btrfs wiki. Available from: `http://btrfs.wiki.kernel.org/` [cited May 2009].

[32] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux Symposium*, pages 21–33, Ottawa, Ontario, Canada, 2007.

[33] Tar - gnu project. Available from: http://www.gnu.org/software/tar/ [cited May 2009].

[34] libarchive. Available from: http://code.google.com/p/libarchive/ [cited May 2009].

[35] Jonathan Corbet. Fibrils and asynchronous system calls. *LWN.net*, January 2007. Available from: http://lwn.net/Articles/219954/ [cited May 2009].

[36] Paul Menage. Cgroups (control groups). *Linux Kernel Source v2.6.29*. Available from: Documentation/cgroups/cgroups.txt.