Imperial College London
Department of Computing

Data Management Strategies for Relative Quality of Service in Virtualised Storage Systems

Felipe Franciosi

April 10, 2012

Supervised by Dr William J. Knottenbelt

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of Imperial College London
and the Diploma of Imperial College London

Declaration

I hereby declare that the research presented in this thesis is my own work, and to my best knowledge it contains no material previously published or written by another person, except where explicitly stated, cited or acknowledged otherwise.

Felipe Franciosi

Abstract

The amount of data managed by organisations continues to grow relentlessly. Driven by the high costs of maintaining multiple local storage systems, there is a well established trend towards storage consolidation using multi-tier Virtualised Storage Systems (VSSs). At the same time, storage infrastructures are increasingly subject to stringent Quality of Service (QoS) demands. Within a VSS, it is challenging to match desired QoS with delivered QoS, considering the latter can vary dramatically both across and within tiers. Manual efforts to achieve this match require extensive and ongoing human intervention. Automated efforts are based on workload analysis, which ignores the business importance of infrequently accessed data.

This thesis presents our design, implementation and evaluation of data maintenance strategies in an enhanced version of the popular Linux Extended 3 Filesystem which features support for the elegant specification of QoS metadata while maintaining compatibility with stock kernels. Users and applications specify QoS requirements using a `chmod`-like interface. System administrators are provided with a character device kernel interface that allows for profiling of the QoS delivered by the underlying storage. We propose a novel score-based metric, together with associated visualisation resources, to evaluate the degree of QoS matching achieved by any given data layout. We also design and implement new inode and datablock allocation and migration strategies which exploit this metric in seeking to match the QoS attributes set by users and/or applications on files and directories with the QoS actually delivered by each of the filesystem's block groups.

To create realistic test filesystems we have included QoS metadata support in the Impressions benchmarking framework. The effectiveness of the resulting data layout in terms of QoS matching is evaluated using a special kernel module that is capable of inspecting detailed filesystem data on-the-fly. We show that our implementations of the proposed inode and datablock allocation strategies are capable of dramatically improving data placement with respect to QoS requirements when compared to the default allocators.

Acknowledgements

I would like to thank the following people:

- My supervisor, Dr William Knottenbelt, for selecting me for this project and for being so helpful, encouraging and constantly optimistic with every achievement during my research.
- The Head of the Department of Computing at Imperial College London, Prof Jeff Magee, for the financial assistance at the start of my studies.
- The people at the Analysis, Engineering, Simulation and Optimisation of Performance (AESOP) research group. In particular: Prof Peter Harrison, Dr Jeremy Bradley, Dr Tony Field, Dr Nicholas Dingle, Dr Abigail Lebrecht and Dr David Thornley.
- The DoC Computing Support Group (CSG) for the countless promptly provided help in so many situations. In particular: Duncan White, Geoff Bruce, Dr David McBride, Lloyd Kamara and Alex Yip.
- The Engineering and Physical Sciences Research Council (EPSRC) for providing me with the funding to do my PhD.
- And last but not least my family and friends, in the UK and abroad, for their love, support and understanding especially during the hard times and events that took place after I left my home country.

This thesis is dedicated to the loving memory of
my godfather Prof Dr Roberto Corrêa Chem,
my godmother Vera Dolores Mainieri Chem,
my cousin Letícia Mainieri Chem
and the other 225 passengers and crew members
of the Air France flight 447 that disappeared
over the Atlantic Ocean on 1 June 2009.
May you all rest in peace.

Contents

List of Figures

List of Tables

1. Introduction

1.1. Motivation

For many decades, the world has witnessed a digital data explosion that shows no signs of abating. This has been partially driven by a dramatic reduction of the cost per gigabyte of storage, which has fallen since July 1980, from the order of US$ 200 000 [45] per gigabyte to less than US$ 0.10 per gigabyte today [52]. It has also been driven by the rise of the use of digital technologies which are now replacing their analog counterparts in almost all environments. Only in 2009, despite the economic slowdown, the IDC reported that the volume of electronic data stored globally grew by 62% to 800 million terabytes (or exabytes). This surge in data volumes is anticipated to continue; indeed by 2020 it is expected that there will be 35 billion terabytes (or zettabytes) of data to manage [82].

In the face of this data explosion, classical storage infrastructures involving multiple local storage systems quickly become difficult to manage, hard to scale and ineffective at meeting evermore-stringent Quality of Service (QoS) requirements as dictated by business needs. These pressures have led to a well established trend towards storage consolidation as typified by the deployment of multi-tier Virtualised Storage Systems (VSSs).

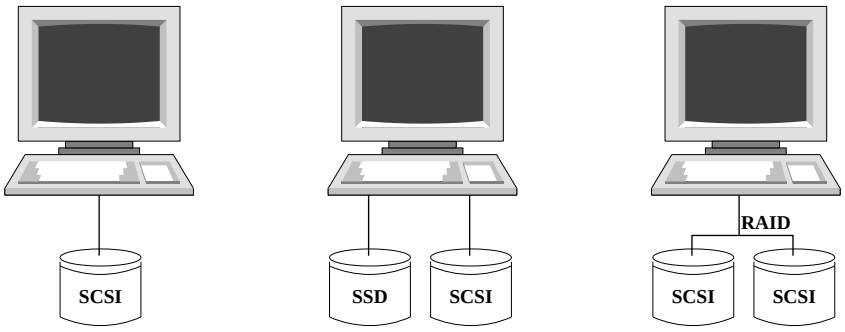


Figure 1.1.: Three examples of Direct-attached Storage (DAS) setups.

To clarify this scenario change, we first present Figure 1.1 illustrating some classical storage infrastructures, which are also known as Direct-attached Storage (DAS). While some of them are as simple as one or more local disks (of the same type or not) attached to a host, others may involve setups such as Redundant Array of Inexpensive Disks (RAID) in order to aggregate two or more disks.

On account of the constraints imposed by DAS, dedicated storage infrastructures took their place in the market. This approach allowed data centres to create centralised storage pools, providing several benefits. Firstly, it achieved management flexibility by allowing storage administrators to arrange media in a single place, facilitating and lowering the cost of data replication, backups and providing improved performance. Secondly, space efficiency was naturally augmented, considering that disk space not in use by one host could be easily allocated to another. Thirdly, scalability also improved due to the ease of adding new media to such platforms. These infrastructures are illustrated in Figure 1.2 and can be classified either as Network-attached Storage (NAS) or Storage Area Network (SAN).

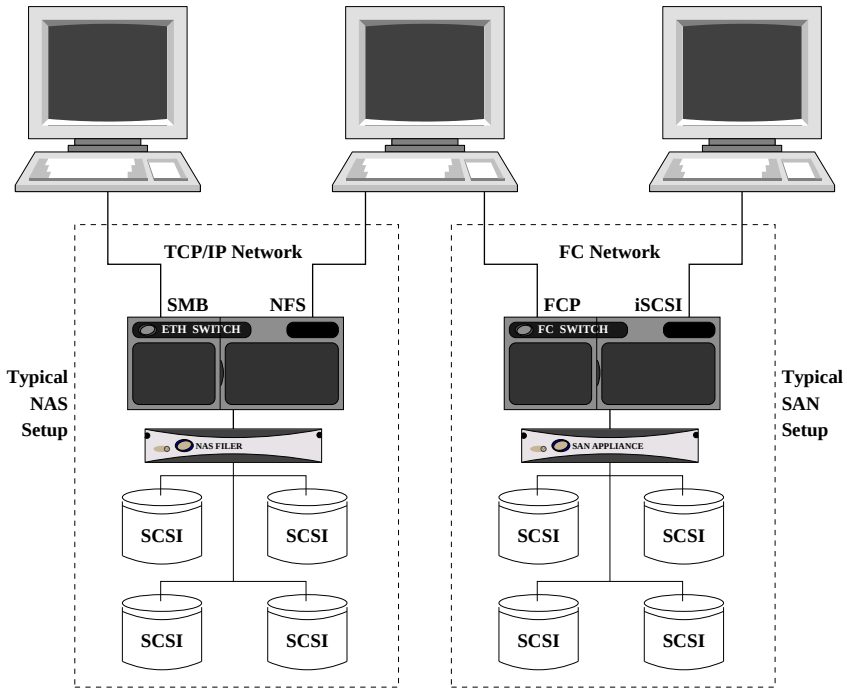


Figure 1.2.: Illustration of two centralised storage solutions: Network-attached Storage (NAS) and Storage Area Network (SAN).

NAS aims to provide heterogeneous hosts with storage access at a *file* level, usually over protocols such as Server Message Block (SMB) or Network File System (NFS). Because it uses network protocols, it usually operates via Internet Protocol (IP) networks and makes the resources visible to the hosts as external media. Its counterpart, SAN, is a dedicated storage network usually implemented over Fibre Channel Protocol (FCP), Fibre Channel over Ethernet (FCoE) or Internet SCSI (iSCSI). It aims to provide hosts with *block level* access to the storage resources, which are viewed as local devices (even when implemented over Ethernet).

With these dedicated appliances providing greater scalability, smaller total cost of ownership (TCO) and ease of management, the concept of storage virtualisation emerges, aggregating the abstraction layer that takes place between the host and the actual storage pool. In the same fashion as RAID arrays (which may also be considered virtualised storage), the host is not concerned with the exact composition of the underlying storage media but instead sees a single logical storage pool.

To conclude the scenario change, we arrive at the aforementioned multi-tiered VSS. It consists of a virtualised storage pool that is composed of different tiers, each delivering different QoS characteristics. As an example, Figure 1.3 illustrates a scheme where a host accesses what appears to be a single logical volume. The volume is being delivered by a SAN appliance and is composed of a series of RAID arrays, each of which possesses different QoS characteristics.

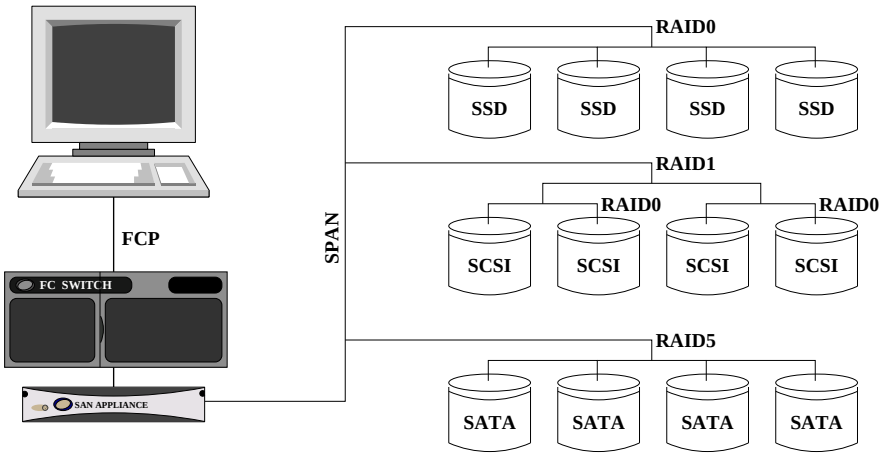


Figure 1.3.: Illustration of a multi-tiered Virtualised Storage System (VSS).

In such a typical scenario, the host's operating system (OS) identifies a single logical drive that would actually be composed of the *span* of three different tiers. The first tier consists of four Solid State Disks (SSD) simply striped as RAID0. The second tier is composed by two pairs of SCSI disks (striped as RAID0), mirrored via RAID1. The third and last tier has four SATA disks using distributed parity, in the fashion of RAID5. Each of these tiers consumes a different amount of power, possesses a different cost per GB of data and delivers different QoS characteristics. The details, pros and cons of different RAID levels will be discussed in detail in Section 2.1.

While VSSs are effective at addressing management overheads and increasing scalability, the issue of ensuring QoS remains an important concern, especially since QoS can vary dramatically both across and within tiers of the same virtual storage pool. Manual data placement and reorganisation are very high overhead activities, especially in the face of rapidly evolving requirements and data access profiles. The practical realisation of a framework for the automation of such tasks has at least two fundamental prerequisites. The first is the ability for users to easily and explicitly specify the QoS requirements of their data in terms of factors such as performance and reliability. The second is a mechanism for mapping those requirements onto low level data placement and migration operations.

This view was supported by Steve Kleiman, Network Appliance's Senior Vice President and Chief Scientist, who is responsible for setting future directions for the company. In his keynote address at FAST 2007, Kleiman defined datasets as "a collection of data meaningful to the user or data administrator having similar properties". Furthermore, he affirmed that these datasets have properties such as QoS and that it needs to ensured that the "right decisions are made by the right people", supporting the idea that users and applications should specify dataset's requirements and storage administrators should find means to deliver it [50]. Kleiman then concludes the presentation of his vision with ideas on simplification through integrated data management and how this should facilitate the management process, eventually setting a goal of 80% to the automation of the workflow. Our interpretation on his view and ideas are expressed as the objectives of this thesis and listed in the following section.

1.2. Aims and Objectives

The primary objective of this thesis is to introduce and explore the concept of matching desired QoS hints from datasets with the actual QoS delivered by a multi-tiered virtualised storage infrastructure. This should be supported by related strategies for data layout evaluation (QoS-wise) and techniques for data placement and migration.

In order to fulfil this, the following objectives must be achieved:

- Development of elegant and straightforward mechanisms for QoS specification. This involves the desired QoS specified by users or applications and the QoS delivered by the storage infrastructure specified by a storage administrator.

- Development of a data layout evaluation strategy to assess the degree of matching between QoS desired and QoS delivered. This involves the creation of mechanisms, configured as a formula, for calculating the score of a given data layout.

- Development of QoS-aware data allocation and migration algorithms. This should utilise the strategies from the previous item in order to make intelligent data allocation and to migrate data in order to improve the layout score of a given filesystem.

- Implement the proposed framework in a working prototype. In order for this prototype to be correctly assessed and easily deployed, it should be usable in real world scenarios and compatible with existing technologies.

- Implement support tools for the visualisation of data layout quality QoS-wise. Ideally, this should provide means for the evaluation of different QoS attributes individually.

- Using the prototype, evaluate the developed allocation and migration algorithms employing a reliable benchmarking mechanism. This involves comparing data layouts after using default allocators, QoS-wise allocators and the migration mechanisms.

1.3. Contributions

This thesis approaches the problem of data placement in multi-tier VSSs from an innovative angle. Current solutions to ease the management burden of such storage infrastructures rely mostly on workload analysis and QoS inference, aiming at improved overall average system performance.

We propose a new approach, where users and applications can provide hints regarding the desired QoS of particular datasets. In the meantime, storage administrators can profile the storage infrastructure regarding the QoS delivered. Allocation algorithms that take into consideration these two parameters together with the current allocation scenario are capable of improving data layout QoS-wise.

In the research literature, there have been many works, both from academic and industrial perspectives, that have proposed QoS frameworks and policies that show the theoretical benefits of an automated approach to QoS management in storage infrastructures [13, 99, 7, 9, 4, 42, 65]. Figure 1.4 illustrates the principle adopted by some of these solutions. It shows a traditional SAN environment where a conventional virtualised storage optimisation system loops over four steps to attain an improved overall system performance. Firstly, it analyses data workflow in terms of access types, frequencies and sizes. Secondly, it hence infers a set of QoS requirements (which may differ from actual needs). Thirdly, it plans a different data layout that would better serve the inferred QoS. Finally, it modifies the data layout, usually by migrating data, prior to looping back to the first step.

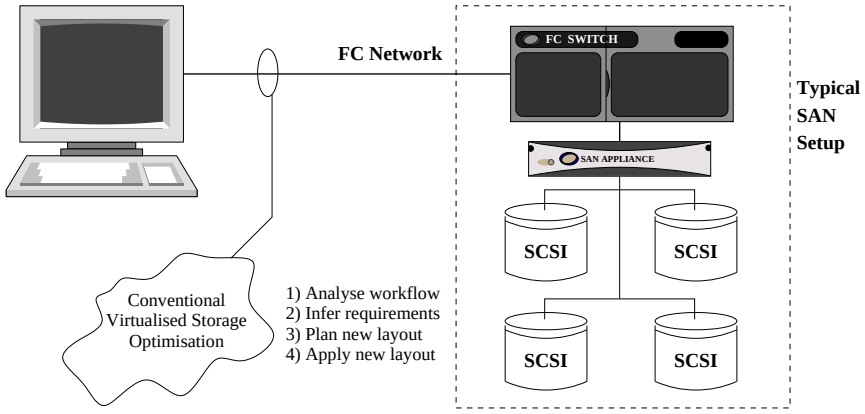


Figure 1.4.: A typical approach to improve overall system performance.

1.3.1. Example State of the Art

To explain in further detail how other solutions attempt to tackle the issue discussed in this research, we describe one popular and recent related work in more depth. BORG [13], short for Block-reORGanization for Self-optimizing Storage Systems, consists of a framework to maintain a cache structure on-disk according to constant data workload analysis.

This cache aims to keep frequently accessed data close together on the disk, minimising seek times and rotational delays during both read and write operations. A background process stays responsible to populate the cache according to workload analysis realised between the filesystem level and the I/O scheduler for the underlying device. In the same fashion, the process operates a write-back policy to flush the write buffer back to the correct position on the disk, outside of the BORG partition.

The work was evaluated according to a Linux implementation of the framework. It was tested under a variety of workloads, including a standard desktop environment, a developer's workstation, a SVN server and a web server. In all the listed cases, BORG provided an improved overall performance ranging from 6% to 50%, being especially notable on cases of non-sequential writing.

Figure 1.5 illustrates the BORG system architecture, indicating the area of the storage stack that the kernel implementation is placed, as well as pointing out the user space components. Analogous to the research done during the elaboration of this thesis, BORG authors believe an implementation at the upper layers such as the VFS prevents the optimisation software to access information regarding physical block addresses. On the other hand, working directly on the device driver level is also complicated due to the lack of logical information such as process IDs and timestamps, which are usually lost during I/O scheduling.

BORG finds itself between the filesystem layer and the I/O Scheduler, aiming for a generic solution that is filesystem independent. It works by initially profiling I/O workloads and passing the information to a user-space analyser and planner. New data layouts for the BOPT (BORG OPtimised Target) partition is passed back to a reconfigurator in kernel space, which in turns maintain data in the cache accordingly. At all times, an I/O Indirector works as a mapper to rewrite read and write requests to point to BOPT.

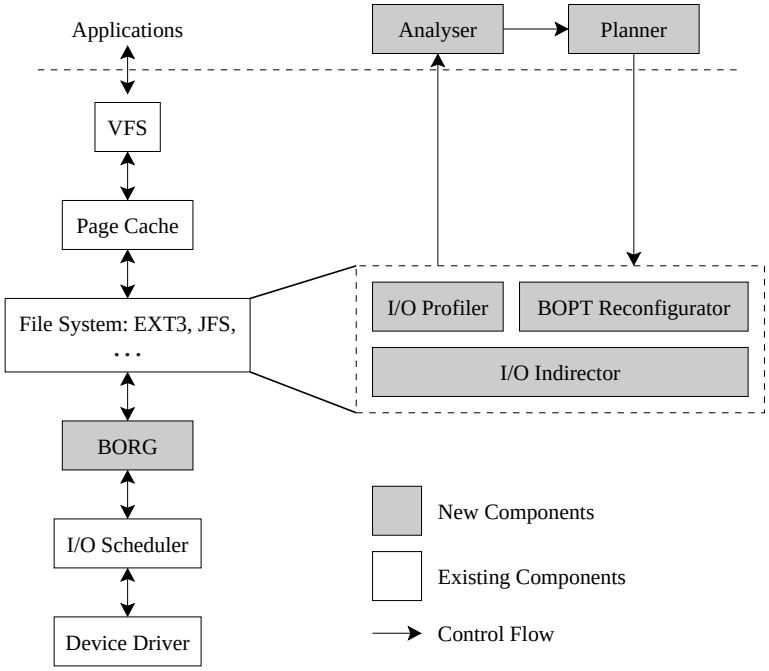


Figure 1.5.: BORG System Architecture [13].

It is interesting to note that the authors have explicitly decided to copy data into the cache (or the BOPT, in this case) maintaining the position that they lie in the filesystem instead of shuffling them in a different order. A shuffling technique could easily be applied to deterministic workloads such as application start-up or content indexing, thus optimising read performance.

However, the authors comment on the fact that some deterministic non-sequential accesses may only be a temporary phenomenon, suggesting that the data is likely to be accessed again at some different point in time on the same order as they lie in the filesystem. This observation is supported by Akyurek and Salem [4], who have argued in favour of copying rather than shuffling [80, 96].

Finally, while BORG was initially tested for single-disk systems, its concepts can easily be applied to dedicated volumes and on top of other strategies listed in the related work. Nevertheless, BORG is clearly classified with the solutions that attempt to automatically improve overall system performance by analysing workloads and inferring access patterns.

1.3.2. A New Approach

The primary contribution of this work is, for the first time, to realise a practical and usable QoS framework within the context of a widely used commodity filesystem, namely the Linux Extended 3 Filesystem (`ext3fs`). This is the default filesystem for many Linux distributions, and consequently has a very large installed user base. `ext3fs` will be discussed in detail, alongside other filesystems, in Chapter 3.

There are several advantages to our realised enhancements to `ext3fs`. Firstly, specification of QoS requirements is as simple as modifying file and directory permissions with a `chmod`-like command; this is done by adding QoS metadata to unused inode space. Secondly, the enhanced filesystem is completely forwards and backwards compatible with existing `ext3fs` installations and vice versa; that is, filesystems may be mounted either way without the need for any conversion. Thirdly, we have implemented mechanisms which allow for the on-the-fly evaluation of the desired and delivered QoS levels; this is supported by a suite of visualisation tools.

To maintain simplicity and elegance within our approach, we do not support the specification of complex absolute QoS requirements such as "95% of I/O requests to file F_1 must be accomplished within 10 ms" or "the file F_2 must be stored with an availability of over 99.99%". Instead, we provide support for combinations of simpler relative QoS goals such as "the file F_1 should be stored in one of the most highly performant areas of the logical storage pool" or "the file F_2 must be placed in an very reliable storage location". Similarly we allow for system administrators to easily specify the relative levels of QoS delivered by different range of block groups within the pool; this process is supported by automated performance profiling tools.

Our approach, illustrated in Figure 1.6, also differs from solutions that attempt to automatically deliver high levels of performance by analysing frequency and type of data accesses [13, 99, 7, 4, 42]. These systems not only ignore other aspects of QoS such as reliability (usually assuming that the entire pool is reliable), space efficiency and security, but also do not cater for the case where mission-critical data is infrequently accessed (and therefore could be profiled as low in performance requirements). This last example would be the case where a relational database table is infrequently accessed but part of a transaction that, when executed, must conclude as

rapidly as possible. While Figure 1.6 illustrates our approach using *low*, *medium* and *high* QoS levels, any numerical score could be used in practice.

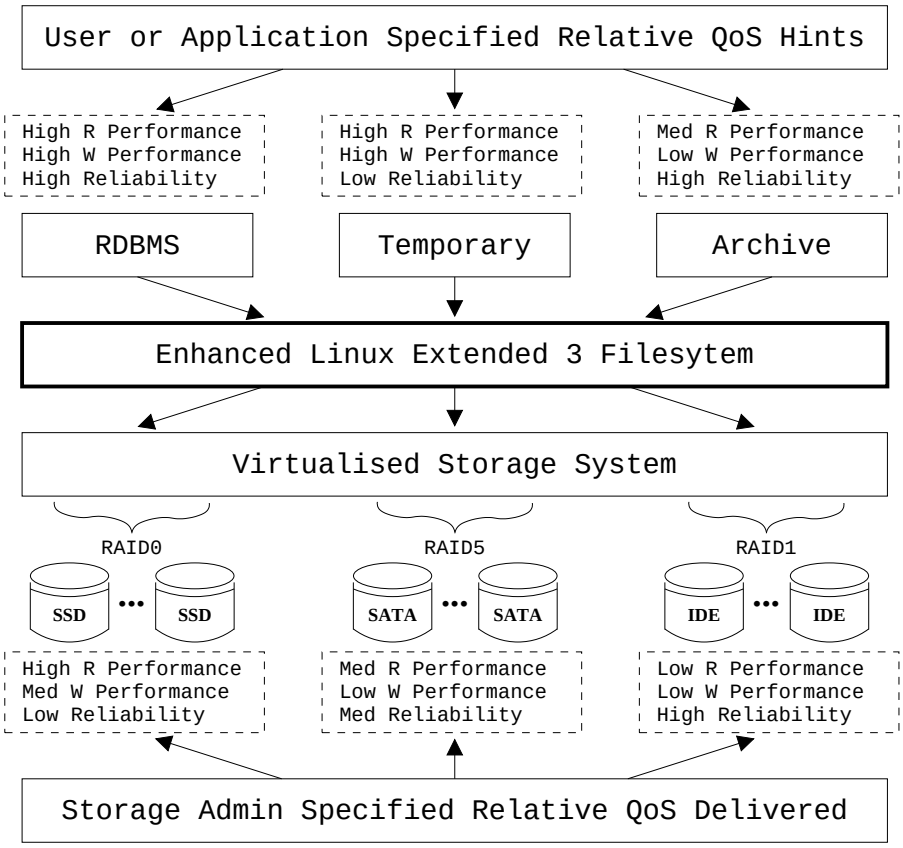


Figure 1.6.: Overview of our proposed approach, where QoS hints are provided by a user and matched through an enhanced filesystem fabric to locations within a virtualised storage system profiled by a storage administrator.

1.4. Further Context

This research takes place in the context of the EPSRC funded proposal EP/F010192/1, entitled Intelligent Performance Optimisation of Virtualised Data Storage Systems (iPODS) [51]. The proposal raises two key questions from the rapid data growth observed in Section 1.1. Firstly, it asks how best to map data onto physical disk devices. Secondly, it seeks to establish the factors on which to base the choice of this mapping.

To answer these questions the proposal focuses on multi-tier Virtualised Storage Systems. As also discussed in Section 1.1, each tier delivers a different cost/capacity ratio against a certain Quality of Service. Examples of such attributes are performance, reliability and power consumption. Due to the usage of multi-zone disk drives and configurations such as RAID, both discussed in Chapter 2, these variations also occur within these storage tiers.

Part of the main objectives listed on the proposal focused on numerical and analytical analysis of such storage infrastructures. Within this scope, both simulations and Markovian models were created to study the internal behaviour of multi-zone disk drives and the usage of such disks in RAID arrays and virtualised storage [53, 54, 55, 56, 57, 58, 59, 97]. Still within this scope, studies modelling the response time [41] and wear levelling [100] of flash drives were also conducted.

Another part of the proposal also listed main objectives relating to the creation of data placement algorithms and migration policies for moving data between tiers and across devices of the same tier. The present thesis addresses these objectives by developing a more sophisticated fabric intelligence, presented in the form of a filesystem that is able to autonomously and transparently allocate and migrate data as solicited by the proposal. This is achieved while observing the Quality of Service attributes required by datasets and the capabilities delivered by the storage infrastructure.

Meeting further proposal requirements, this work introduces the means to quantitatively assess the benefits of the newly presented algorithms. This is accomplished in two ways. Firstly, a numerical score of the data layout evaluation which can be compared to other setups. Secondly, layout bitmaps that instantly provide a perspective on the algorithm's performance by displaying how accurately required and delivered QoS attributes are matched.

1.5. Publications

The following publications arose from work conducted during the course of this PhD:

- **UK Performance Evaluation Workshop 2009** (UKPEW) [34] presents the first steps towards the concrete realisation of a filesystem that supports QoS metadata. It also introduces our data layout evalu-

ation scheme and the first thoughts towards allocation and migration algorithms. Material from this paper appears in Chapters 2, 3 and 4.

- **23rd Annual European Simulation and Modelling Conference** (ESM) [15] introduces techniques to explore the viability of storage systems where data is placed intelligently depending on data-specific QoS requirements and the characteristics of the underlying storage infrastructure. The studies also evaluate automatic profiling techniques and simulation of data stream workloads.

- **27th IEEE Symposium on Mass Storage Systems and Technologies** (MSST) [35] presents a functional QoS-aware filesystem, using a more elaborate version of the data layout evaluation formula previously introduced. Intelligent allocation strategies making use of such evaluation mechanisms are also implemented and benchmarking is done using an extended version of Impressions [3]. Material from this paper appears in Chapters 4 and 5.

An additional paper including extended experimental results is in preparation for submission to **The Computer Journal**.

2. Multi-tier Virtualised Storage Systems

The definition of virtualised storage is broad and may be used to describe any sort of storage system where a host is incapable of identifying the exact media details comprising the infrastructure at hand. Considering that, the term may be applied, for example, to refer to a RAID array located inside a computer, a NAS enclosure connected to a TCP/IP network or even a SAN enclosure connected through a dedicated Fibre Channel network (as discussed in Chapter 1).

This work, however, will allude to a Virtualised Storage System (VSS) when referring to an external enclosure, presented as a SAN, in the form of a centralised storage infrastructure that is perceived by the host system as a local logical volume. This category of equipment includes (but is not limited to) a series of hard disk drives, usually attached through controllers in the fashion of RAID arrays.

The host system will therefore have access to the storage pool at block level, instead of file level as is the case with NAS. This is a fundamental difference considering the fashion we choose to approach the problem of placing data with regards to QoS parameters, as some of our employed techniques would be limited if we did not have access to information such as the Logical Block Addresses (LBA) of the logical volume.

Inside the VSS, each collection of similarly arranged storage devices can be classified as a tier. Considering that different types of collections may be provided, either by the usage of distinct storage media or simply by rearranging them in the controllers that they are connected to, different benefits such as performance, reliability and space efficiency are yielded. An enterprise storage system must therefore provide a multi-tiered solution, since its applications will most likely have different requirements.

2.1. Tier Organisation by RAID Level

In [92], it is suggested that an enterprise should define different storage Service Level Agreements (SLAs) for its applications on five different tiers, based on the requirements imposed by each application. This work, however, simplifies this definition to use tiers based on the four most commonly used RAID levels. The following subsections describe each one of them and are based on [76] unless noted otherwise.

2.1.1. RAID0 - Striped Volume

The RAID0 configuration is composed of a set of disks where blocks are striped across the disks. Because there is no redundancy whatsoever, any disk failure will result in data loss. The benefits of such approach are the performance and space efficiency gains. Both read and write performances are improved due to the RAID controller being capable of breaking down I/O operations and realising them over more than one disk at once. The space efficiency gains occur because no space is dedicated to data parity or replication, sacrificing reliability. Figure 2.1 illustrates the configuration.

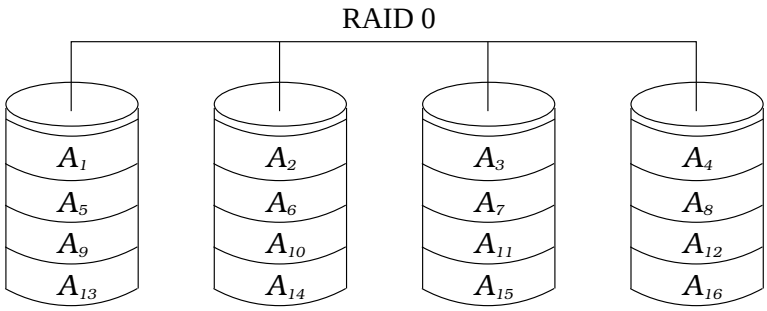


Figure 2.1.: RAID 0 - Data is striped over several disks.

In the above example, each block of data is illustrated as A_i. These are RAID blocks, whose size is dictated by the stripe size, and should not be confused with a filesystem datablock. They are distributed so that consecutive data reside on different disks; this being the key for the striping of I/O operations aiming for performance gains. The gains will also be dependent on the size of the operation (that is, they must be over one block) and the size of the blocks themselves, which is the same over the entire array and which is configurable by the RAID controller.

2.1.2. RAID5 - Striped Volume With Distributed Parity

The next configuration we consider is RAID5. Similar to RAID0, data is striped across a set of disks. For every set of blocks, however, a parity is calculated and distributed in a rotating fashion among the set as shown in Figure 2.2. When compared to the less popular RAID levels 3 or 4, where the data parity is stored on a dedicated disk, RAID5 provides better performance by eliminating the bottleneck imposed by the dedicated disk, in view of the parity update required by any write operation to any disk in the array. On the other hand, in the case of a disk failure, RAID5 will still operate in degraded mode until the failed disk is rebuilt on a spare.

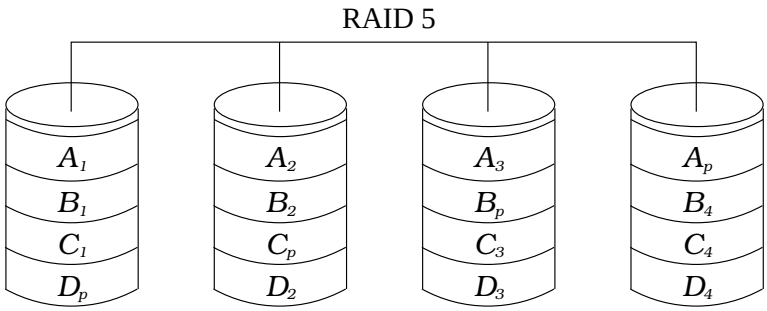


Figure 2.2.: RAID 5 - Striped set with distributed parity.

2.1.3. RAID6 - Striped Volume With Double Distributed Parity

The RAID6 configuration appears identical to RAID5, but uses two parity blocks (usually named P and Q) instead of one. However, the exact implementation of the parity scheme may vary. According to the Storage Networking Industry Association (SNIA), the definition of RAID6 is "any form of RAID that can continue to execute read and write requests to all of a RAID array's virtual disks in the presence of any two concurrent disk failures" [11]. A possible implementation is illustrated in Figure 2.3.

Due to this vague definition, different vendors have implemented RAID6 in different ways. As an example, IBM presented EVENODD [14], which uses an exclusive-OR based parity scheme instead of Reed-Solomon error-correcting codes [77]. Another example is a Network Appliance's technique called Row-Diagonal Parity [23] which also uses exclusive-OR parity.

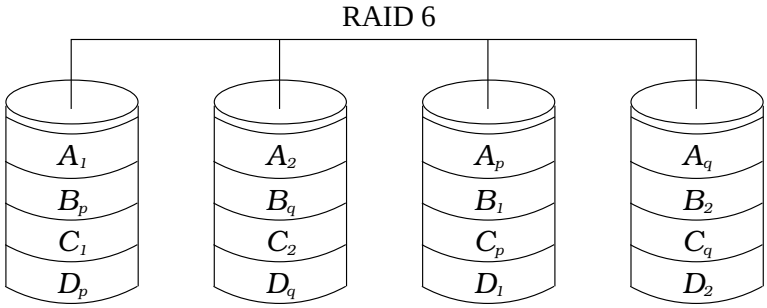


Figure 2.3.: RAID 6 - Striped set with distributed double parity.

The main motivation for the creation of such a configuration was the probability of data loss during the recovery process upon a first disk failure in the array. Given that the reliability of hard disk drives does not increase in the same pace as their capacity, larger RAID5 arrays will be more likely to sustain an unrecoverable disk error during the recovery process [21]. RAID6 improves on that by supporting a secondary drive failure.

2.1.4. RAID10 - Striped Mirrored Volume

The RAID10 configuration is composed by nesting RAIDs level 0 and 1 together. While RAID0 provides a striped set and RAID1 provides mirroring, RAID10 offers a RAID0 array of a mirrored set of disks, as shown in Figure 2.4. While this model supports multiple drive failures, as long as no two drives fail in the same mirror, some space penalty is imposed by the redundant data present in the array. The performance gains or losses are highly dependent on implementation details.

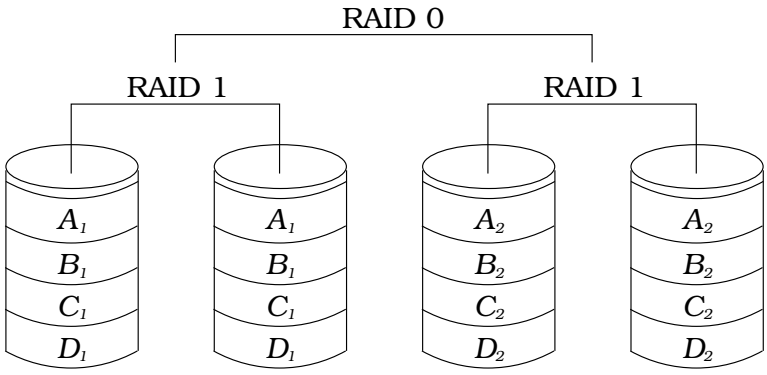


Figure 2.4.: RAID 10 - Striped array of a mirrored set of disks.

Comparing all configurations, we note that the performance of RAID10 and RAID5 may vary depending on the size of reads or writes being executed. That will happen because RAID5 requires the parity in the array to be updated with every write, thus enforcing at least one more read if not every block in the stripe is being updated. In [54], these details and differences are measured and analytically modelled, allowing an approximation to the response time of I/O requests to be predicted. Since RAID6 is not supported by the hardware used in our experiments and the lack of a standard software solution, we have used RAID0, RAID5 and RAID10.

2.2. Usage of Multi-zone Drives Within RAID

Managing the placement of data in multiple tiers that present distinct characteristics is a complicated task. This task does not get any easier when tiers are composed by storage media with attributes such as those of multi-zone hard disk drives, that provide a larger number of sectors on different parts of the platters, and which therefore have different performance levels depending on the block address accessed.

The usage of disks such as Solid State Drives (SSDs) creates a different set of challenges that can be more easily addressed. It is possible to say that SSDs have a constant (and nearly zero) seek time, as well as a constant throughput throughout different addresses. The write speed, however, may suffer for small operations due to the physical organisation of the drive, that usually requires a whole cell to be rewritten even for the smallest data modification. In light of these elements, we would recommend that SSDs were grouped in a separate, exclusive RAID group and considered as a separated tier from other drives with different characteristics.

Magnetic hard disk drives, on the other hand, contain a collection of platters that spin around a spindle. This allows for a mechanical arm to move across the top of each platter and reach the entire media. The arm positions a head capable of accessing data through read and write operations [81].

Originally, these platters were divided into areas called sectors and distributed on the disk as it is illustrated in Figure 2.5. This layout infers that sectors on the edge of the media are bigger in area than those near the centre. Because the angular velocity with which a platter rotates is constant across the media, the time taken to read any two sectors is very similar.

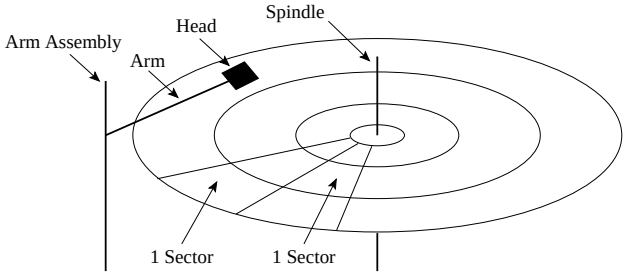


Figure 2.5.: Single-zone disk architecture illustrated with one platter.

This design facilitated the old addressing scheme known as Cylinder-Head-Sector (CHS), which used a combination of parameters to calculate the position of a sector. However, this scheme does not take advantage of two key factors provided by the disk's surface geometry towards its outer edge. Firstly, the linear speed of the disk surface is faster, and that could be used to improve the performance of read and write operations. Secondly, the surface area is bigger, allowing for a greater number of sectors and therefore a larger capacity drive.

With advances in technology and the adoption of a sequential addressing mechanism known as Logical Block Addressing (LBA), hard disk drive manufacturers were capable of increasing the number of sectors per track towards the edge of each platter. Such disks were named *multi-zone* after their varying number of sectors per track. A zone, in this case, would be a collection of tracks containing the same number of sectors. Figure 2.6 illustrates the layout changes in a simplified version of the model. It shows three tracks, three zones and indicates the edge as the "outer zone" containing the greater number of sectors.

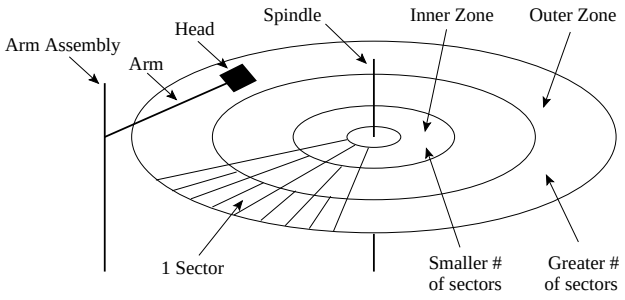


Figure 2.6.: Multi-zone disk architecture illustrated with one platter.

The performance of I/O operations on such drives varies drastically across the disk. To quantify and illustrate these variations, we have written a simple performance measurement utility to evaluate the time taken to sequentially read 100 MB buffers from a 500 GB disk. We profiled a Seagate ST3500630NS multi-zone hard disk drive and plotted the results on Figure 2.7, expressing the read throughput (in MB/s) across the disk.

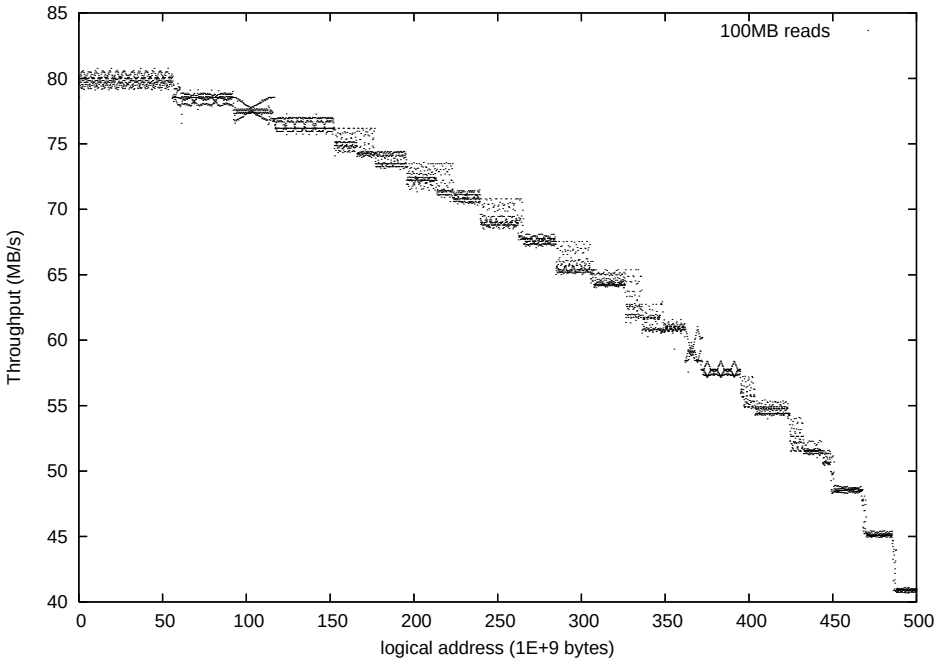


Figure 2.7.: Throughput of a 500 GB multi-zone disk, measured via sequential reads of 100 MB blocks of data.

The plot shows that read operations on the edge of disk (near LBA 0) are almost twice (98%) as fast as those performed near the centre of the disk. This is due to the increased linear speed with which the heads pass over the edge of the platters, thus reducing the transfer time of operations on those sectors. We have also conducted measurements with varying buffer sizes as well as with write operations. For a single drive, however, these parameters did not affect the illustrated throughput significantly.

2.3. Data Placement Within VSSs

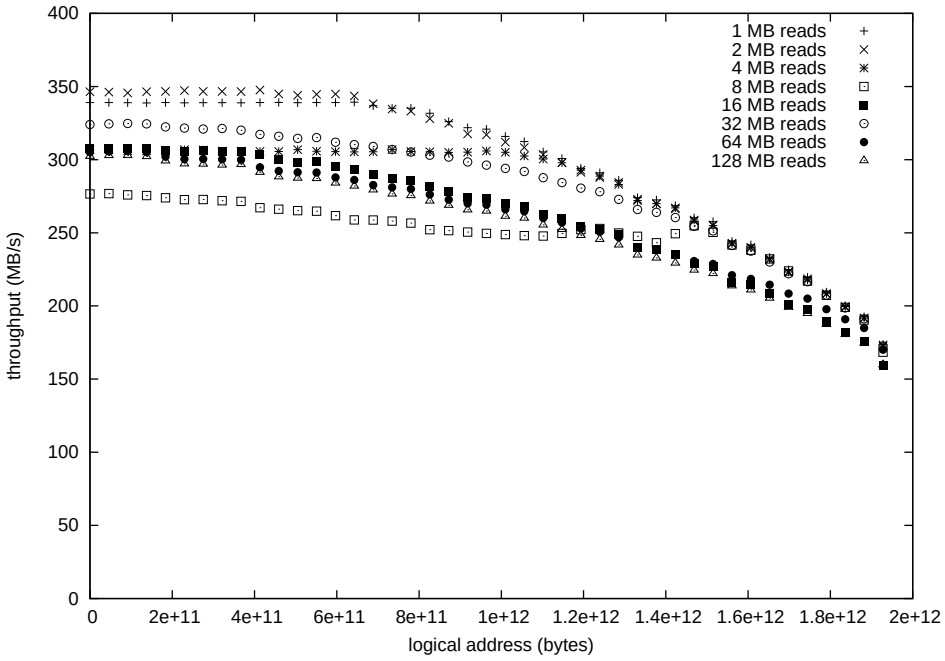
The usage of multi-zone hard disk drives within VSSs brings further complications to the matter of data placement. As shown on the previous section, the I/O performance of such disks will vary considerably depending on the part of the media in use. When they are associated in RAID arrays, the limits of these variations will naturally grow. This will cause situations where parts of an apparently low performance tier provide better throughput than parts of a high performance counterpart within a VSS.

To quantify such variations, we have assembled three different RAID arrays (levels 0, 10 and 5) and profiled them with read and write operations of varying buffer sizes. These RAID levels are popular because they provide good performance (RAID 0), good reliability (RAID 10) and a combination of both (RAID5). The enclosure in use is an Infortrend EonStor A16F-G2430 connected via a dual fibre channel interface to a Ubuntu 7.04 (kernel 2.6.20.3) Linux server with two dual-core AMD Opteron processors and 4 GB RAM, thereby characterising a SAN environment. Each array consists of four multi-zone Seagate ST3500630NS hard disk drives.

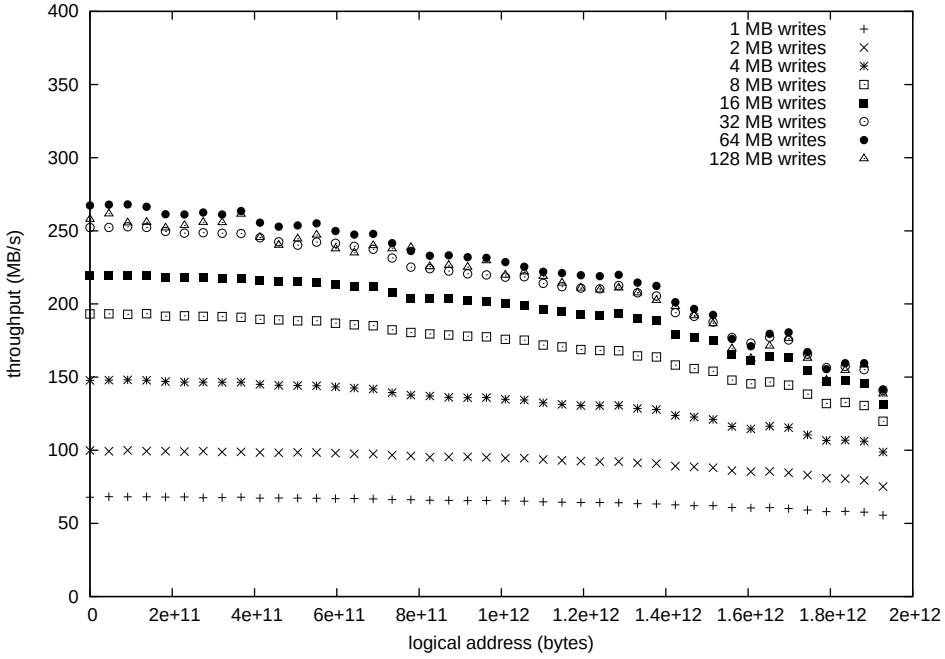
Naturally, the address space of RAID10 and RAID5 will be smaller than that of RAID0 due to data replication (as discussed in Section 2.1). With RAID5 providing some reliability, 500 GB of data is dedicated to keep parity information. On the other hand, RAID10 assigns 1 TB for mirroring.

Regarding the performance variations, we can observe the RAID0 measurements in Figures 2.8(a) and 2.8(b), which illustrate the throughput of sequential read and write operations respectively. It is possible to note that small writes (using 1 MB buffers) will happen at a rate of around 55 MB/s in the slowest areas. By contrast, similar sized reads (using 2 MB buffers) can be achieved at almost 350 MB/s on the fastest parts of the array.

This confirms that, unlike the single disk scenario, where the fastest zones were about 98% faster than the slowest counterparts (independent of buffer sizes), the impact of multi-zone hard disk drives in RAID arrays is much bigger (up to around 500% on our RAID0 experiment). Such performance variations suggest that we must be careful when classifying different tiers regarding their QoS attributes. Figures 2.9(a) and 2.9(b) exhibit the results for read and write operations in RAID10 respectively. Figures 2.10(a) and 2.10(b) show the throughput for RAID5 in the same manner.

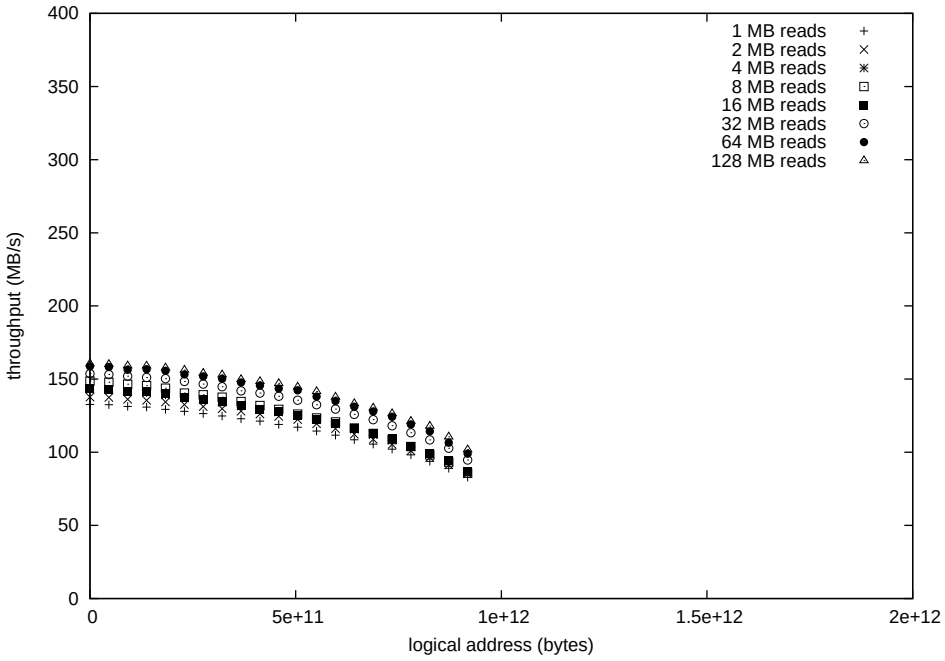


(a) Read Performance

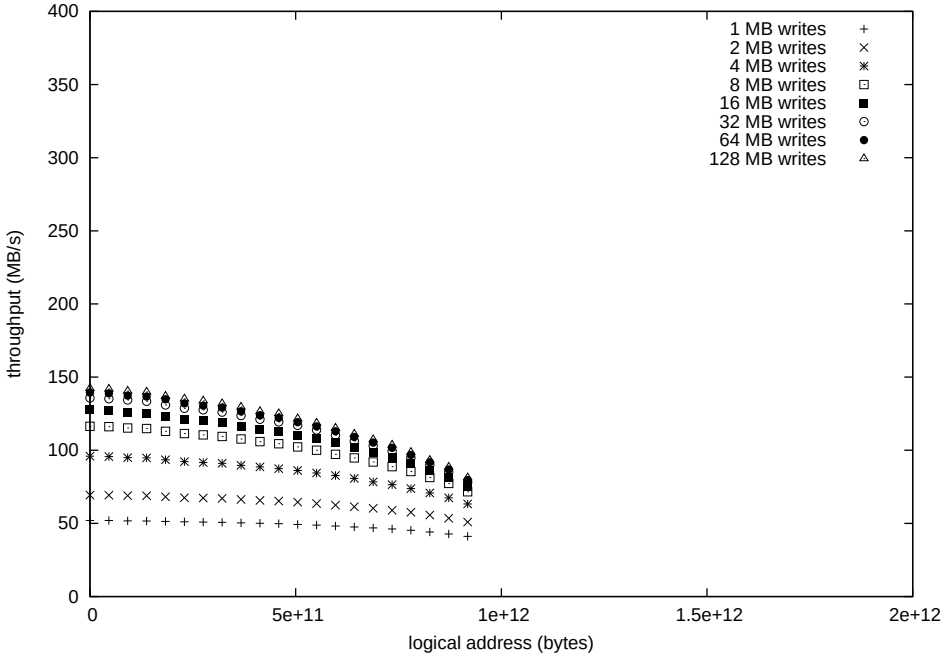


(b) Write Performance

Figure 2.8.: Throughput of a RAID0 array using multi-zone hard disk drives.

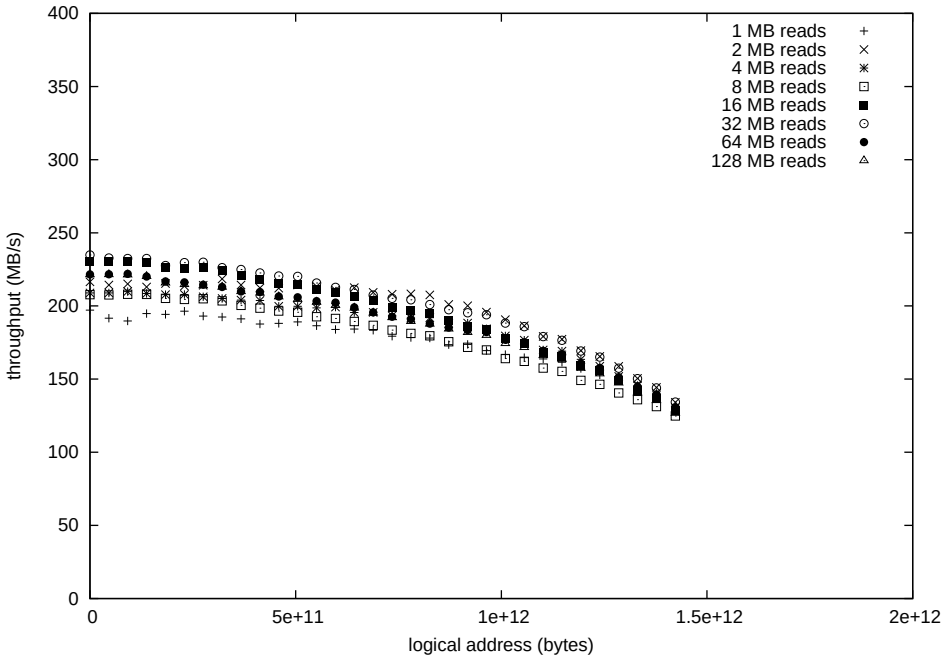


(a) Read Performance

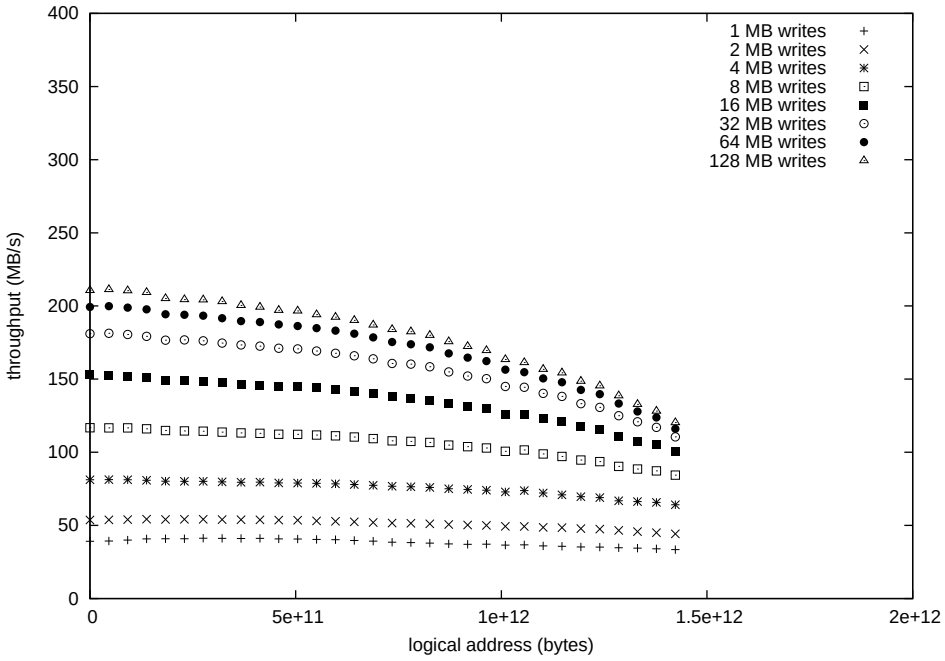


(b) Write Performance

Figure 2.9.: Throughput of a RAID10 array using multi-zone hard disk drives.



(a) Read Performance



(b) Write Performance

Figure 2.10.: Throughput of a RAID5 array using multi-zone hard disk drives.

Prior studies on data placement within multi-zone devices have suggested different approaches for both single disks and RAID arrays. For single disks [37], it is suggested that the zones be identified and their performance attributes mapped. A special filesystem would then keep track of file access profiles and move data that is accessed more often (so called "hot" data) to the fastest areas of the disk.

Regarding multiple disk arrays, Zoned-RAID [49] proposes that a block level approach should be used (instead of a file level equivalent). The concept introduced by the authors improves the performance of traditional RAID levels 1, 5 and 6 by taking advantage of the zone properties in multi-zone devices and arranging blocks appropriately.

A slightly different approach, which has some similar ideas to the ones we have introduced, is proposed in [5]. There, a benchmarking of the system is previously realised and maps are constructed gathering information on the specifics of the disks underneath. While this is not limited to multi-zone devices, these maps are then used to predict the performance of the system at any given logical address.

In order to create a contained framework capable of handling both the distinct QoS attributes of multiple tiers and the QoS requirements of different datasets, we have explored the idea of integrating our scheme into the upper layers of an existing operating system's filesystem. While the exact filesystem to use would depend on the analysis realised in the next chapter, this layer is a natural choice for our work considering it is responsible not only for hosting data such as files and directories, but also for the allocation mechanisms.

3. Filesystems

In order to evaluate and discuss how blocks of data are allocated and used within filesystems, it is important to first understand the structure of such filesystems and how they are organised into a logical volume. This section will first discuss how several storage tiers can be organised into one single logical volume. Next, we explain the Master Boot Record (MBR), which is the structure that lays on the first sector of every partitioned logical drive on the basic disk architecture [18] and then elaborate on three popular filesystems, emphasising their on-disk layout, benefits and drawbacks. Finally, we settle on `ext3fs` as the filesystem to use in this work.

3.1. Single Filesystem Over Several Storage Tiers

While some storage solutions are capable of exporting a single logical volume composed of several different tiers, others will export each tier individually. In order to create a single filesystem over several tiers, unless the filesystem itself supports multiple devices, it is first necessary to assemble the tiers as a single volume.

Using a tool such as the Linux's Logical Volume Manager (LVM) [62], we are capable of assembling a single logical volume that is composed of different devices. This is done in such a way that the logical block addressing starts at the beginning of one tier and increases linearly until the end of the last. Figure 3.1 illustrates this concept.

In order for LVM to run and setup several devices into a single logical volume, at least part of the operating system must be up and running. This means that a "boot partition cannot reside on an LVM volume because the [...] boot loader cannot read it" [78].

However the nature of the volume (boot or data), every volume is usually organised in partitions. The next section describes the MBR, which is a standard for volume partitioning popularised by IBM in the early 80s [86].

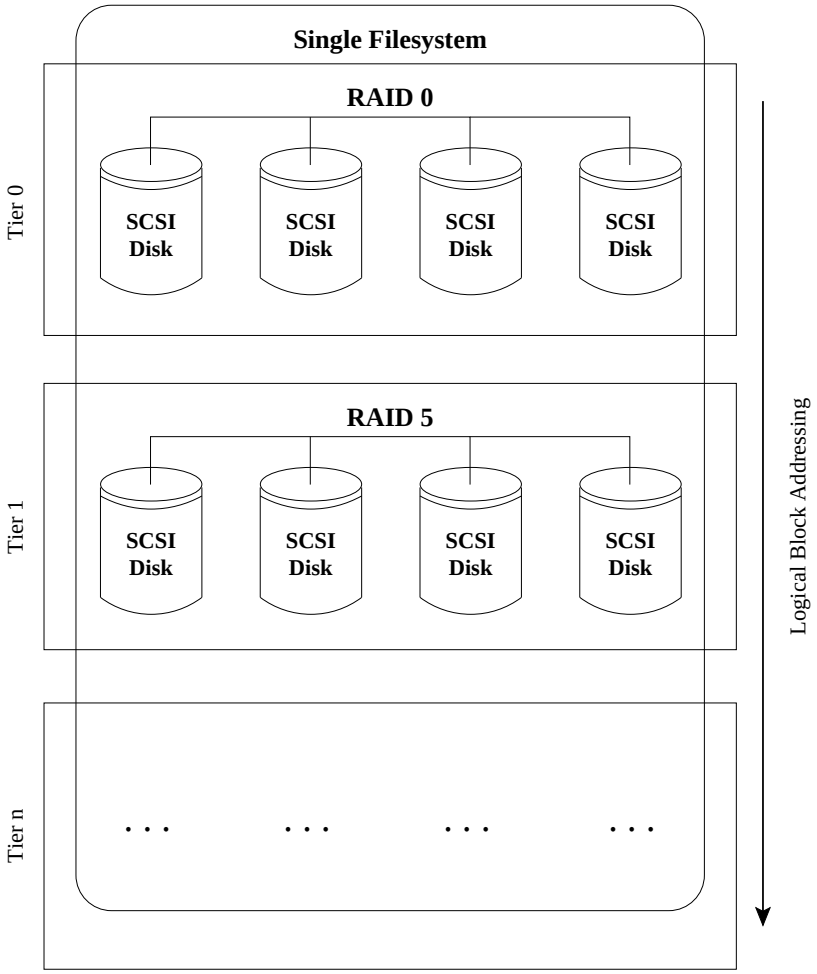


Figure 3.1.: Using Linux's Logical Volume Manager to assemble a single logical volume composed of different devices.

3.2. Master Boot Record

The Master Boot Record (MBR) is a structure that resides in the first sector of every partitioned data storage device in the basic disk architecture. It uses the whole sector (512 bytes) and contains both the instructions to boot up a system and the primary partition table. If the device in question is not used for booting purposes, then only the partition information will be relevant. Figure 3.2 illustrates the structure of the MBR. The remainder of this section is based on [85] unless stated otherwise.

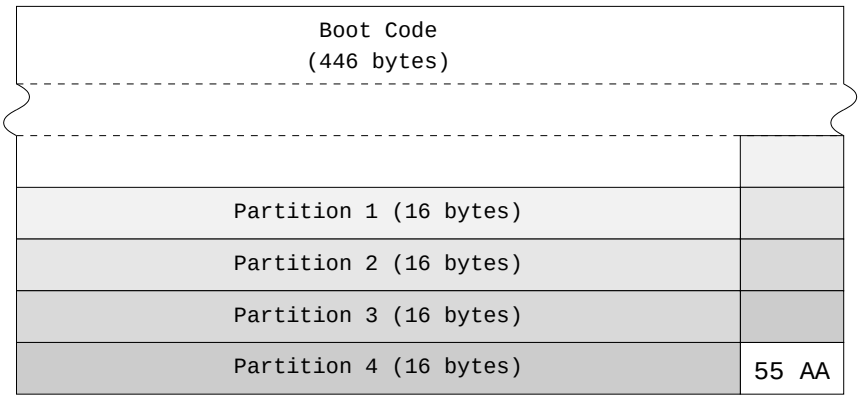


Figure 3.2.: Master Boot Record (MBR) in the basic disk architecture.

As shown in the figure, the first 446 bytes of the MBR is composed of the machine code responsible for booting up the system. On a simple home computer, this is the code loaded by the Basic Input/Output System (BIOS) after performing the Power On Self Test (POST). It is interesting to note that more elaborate (and larger than 446 bytes) boot loaders (such as GRUB [66]) can be loaded by the MBR boot code from some pre-specified special boot partition. The boot loader may also offer a means of loading a different partition table. On a VSS, where the logical volume is usually used simply as a storage media, this part of the MBR can be ignored.

Next to the boot code lies the partition table, occupying 64 bytes. That space is organised as four primary partition records of 16 bytes each. When more than four partitions are needed, the concept of extended partitioning can be used, being the extra partitions defined outside of the MBR. As of today, several different techniques exist for this purpose [26]. The structure of a 16 byte partition record is illustrated in Figure 3.3.

Figure 3.3.: A 16 byte partition record.

The first byte of this record, illustrated in the figure by `BF` (meaning *boot flag*), identifies whether the partition should be bootable or not. The `CHS BEGIN` and `CHS END` fields will contain the position of the beginning and the end of the partition, respectively, addressed using the cylinder-head-sector scheme. The `PT` flag stands for *partition type* and is of particularly interest to this work because it identifies the filesystem type that resides in that partition. Last, the `LBA BEGIN` and `NUM OF SECTORS` fields will identify the logic block address for the beginning of the partition and the number of sectors used by it.

At the end of the MBR, there is also a two byte signature (`0xAA55`) used to suggest that no corruption has occurred. The value `0x55` may appear first on the disk layout because of endianness issues.

While some partitions may be used for the storage of raw data, as is required by certain applications [28], most systems will have them formatted to support a filesystem. The next sections will compare and discuss the structure of two commonly used filesystems, explaining their benefits and drawbacks.

3.3. The FAT Filesystem

The FAT filesystem, named after File Allocation Table, was introduced in the late seventies for the IBM PC machine architecture and incorporated into the first version of MS-DOS shortly after. Because of its simplicity, the filesystem was also used in other operating systems at the time and today it is still used in a series of embedded devices [87]. Its limitations, however, as will be shown later in this section, led to the usage of more complex and elaborate filesystems in newer operating systems. Unless noted otherwise, this section was written based on the Microsoft specifications [24, 27].

The version of FAT today referred to as FAT12 due to the file allocation table consisting of 12 bit pointers, underwent several upgrades over time to support additional features and higher capacity media. Among these

features, it is important to name the support for long file names (bigger than eight characters plus a three character extension) and the increasing size of bits in the file allocation table, making it possible for a single filesystem to be used in larger partitions, once it could address more space.

Despite these upgrades having slightly changed the layout of the filesystem, the basic format has remained the same since the second version of MS-DOS. At the beginning of the filesystem there is a reserved region containing the boot sector (BS) and the BIOS parameter block (BPB), followed by two copies of the file allocation table. Next, data sectors occupy all the space until the end of the filesystem. This format is illustrated in Figure 3.4 without any particular dimensional scale.

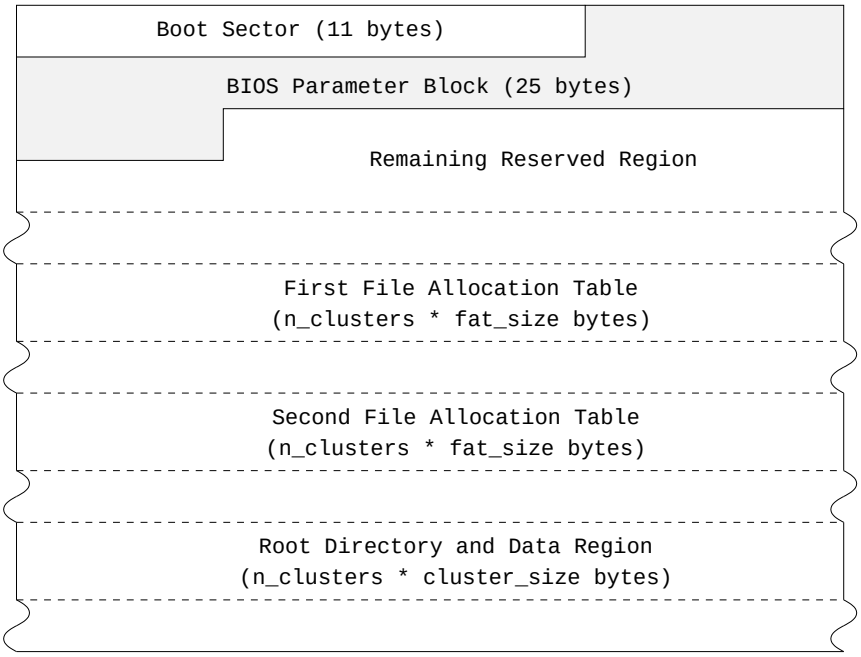


Figure 3.4.: The FAT filesystem layout.

The BS begins with a 3 byte jump instruction which points to executable code located in the remaining reserved region. That is necessary in view of the way the MBR boot loader works; after locating a primary boot partition, it will start executing code at the beginning of that partition. The next 8 bytes in the BS is called Original Equipment Manufacturer (OEM) Name and is usually set to "MSWIN4.1" for compatibility issues, but most operating systems will not pay any regard to this field.

Next is the BPB, which holds crucial information on how the operating system should handle the remainder of the filesystem. This includes aspects such as the FAT version (12, 16 or 32 bits), the size of the reserved region, media description (such as floppy disk or hard drive, including geometry data) and the cluster size. A cluster is a group of consecutive sectors in the storage media and its size plays an important role in the limitations of FAT, as shall be discussed further in this section. More details on the BS and BPB can be obtained in [25].

Up to this point, FAT12, FAT16 and FAT32 shared exactly the same data structures. However, FAT32 introduced a 28 byte BPB continuation which includes extended information and reserved space in an attempt to avoid other structural changes in future versions of the filesystem. Following that, there is a 26 byte volume information field which is shared by all FAT versions, but FAT12 and FAT16 will have this located right after the BPB, since the extended information is not present on these versions.

After the end of the reserved region, which may vary in size, there are two copies of the file allocation table. While one copy would suffice, a second copy has been implemented in an attempt to aid recovery of a filesystem in the event of a sector used by the file allocation table becoming bad. Because old hard disk drives used to develop bad and unusable sectors over time, the occurrence of such misfortune over the allocation table would result in the loss of the whole filesystem.

Before explaining how the file allocation table works and discussing its advantages and disadvantages, it is important to understand the last item illustrated in Figure 3.4, namely the root directory and data region. While the former was made extinct in FAT32 by having been incorporated within the latter as a regular directory, the root directory in FAT12 and FAT16 is a preallocated fixed space for the files in the top level of the filesystem, which is organised as a tree. Disregarding the details on how long file names are implemented, each directory is organised as an array of linear 32 byte records, each record indicating either a file or a subdirectory.

While this 32 byte record holds information such as the file (or subdirectory) name, attributes and timestamps for creation or modification, the relevant field for comprehension of the file allocation table is the one indicating the file's first cluster. This field addresses the cluster in the data region where the data for the file starts. Not only does this allow the operating

system to read the proper sector for the beginning of the file's data, but it also indicates an index for consulting the file allocation table, which will provide the next cluster used by the file or indicate if it is finished.

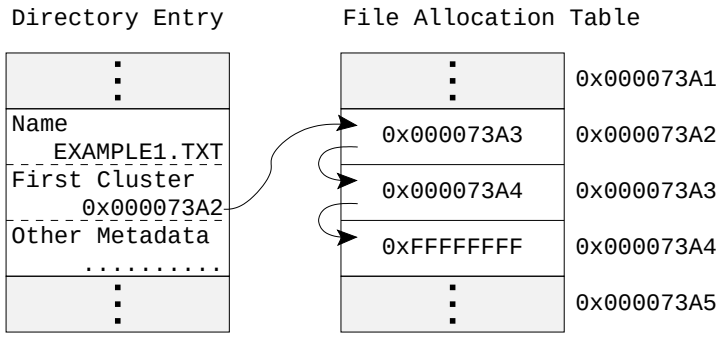


Figure 3.5.: The file allocation table and a contiguously allocated file.

Figure 3.5 shows details of the file allocation table structure. It shows an example of a directory entry, which could be located in the root directory, presenting a file named `EXAMPLE1.TXT` whose first cluster of data resides in the media address `0x000073A2`. Considering that the contents of this file is big enough to occupy three clusters, the file allocation table is then used to find the subsequent clusters. According to the example, this file was allocated contiguously, having the next two clusters of data located at addresses `0x000073A3` and `0x000073A4`. The special address `0xFFFFFFFF` denotes that no clusters are to follow.

However, when a file is deleted from the filesystem, the space for that file is freed for future usage. This implies that when another file is created, it will reuse that same available space. When the new file requires more clusters for its data than is contiguously available, or if a previously allocated file grows, therefore occupying more space, fragmentation may occur. Figure 3.6 illustrates the allocation of a file named `EXAMPLE2.TXT` and how the file allocation table will look like with respect to the data clusters for this file. This also applies for directory entries.

In the figure, it is noticeable that the second chunk of data is allocated in the cluster `0x0094E135`, while the first and second chunks are considerably distant in the storage media, being located at `0x00003CD2` and `0x00004B11` respectively. This kind of fragmentation is likely to cause the file access to be slower if the media being used is, for instance, a hard disk drive.

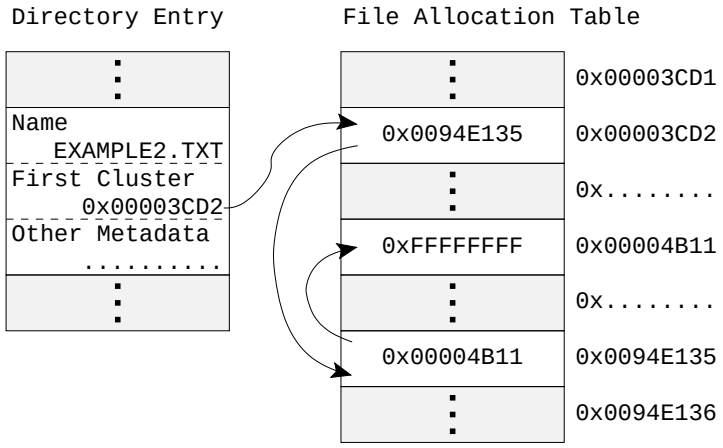


Figure 3.6.: The file allocation table and a fragmented file.

The reason for this increase is that accessing noncontiguous data will cause mechanical drives to move the head of the disk to the proper place (seek time) and rotate the disk to the proper track (latency time).

Properly avoiding fragmentation is still an open challenge in the design of modern filesystems and different techniques have been proposed in an attempt to do so. As will be discussed in Section 3.4, some filesystems reserve disk space for files to grow [16]. Others attempt to delay as much as possible the writing of data to the disk, avoiding fragmentation when several files are written to the disk concurrently [67, 94]. Our research presents a novel technique which will be discussed in Chapter 4.

Regarding these issues, the concept of cluster also plays an important role. In an attempt to support larger partitions, reduce fragmentation and keep the file allocation tables small, FAT will group sectors together in clusters. In doing so, the size of each block of data is actually increased in multiples of the sector size, therefore reserving some space for small files to grow. Naturally, small files that do not grow will occupy space in multiples of clusters, making hard disk free space unusable in the event of a large number of such files. This is also called internal fragmentation.

Nevertheless, the size of the filesystem is still limited by the number of how many clusters the allocation tables can address. In FAT12, filesystems of up to 8 MB are supported when using 4 KB clusters. In FAT16, not only are 4 bits added to cluster addressing, but also cluster sizes of up to 64 KB can be used, allowing for filesystems as big as 4 GB. Microsoft, however,

states that no more than 32 KB should be used as the cluster size, as it will result in compatibility issues with other vendors. The allocation table for FAT32 uses 28 bits for addressing (4 bits have been reserved). Considering the same 32 KB cluster size, filesystems of up to about 8 TB are supported.

3.4. The Extended 2 and 3 Filesystems

The Extended 2 Filesystem (`ext2fs`) is native to the Linux operating system and was based on the UNIX filesystem first introduced by Bell Laboratories in the early seventies [70]. The Extended 3 Filesystem (`ext3fs`) introduced journalling techniques to `ext2fs`, but shares exactly the same structural concepts with no modifications made to it [91]. While journalling techniques aim to allow systems to restart quickly [44], this section will discuss the structural aspects of `ext2fs`. Unless noted otherwise, this section was written based on [16].

In contrast to the FAT filesystem discussed in the previous section, `ext2fs` was designed for operation on hard drives and not floppy diskettes. Considering that hard drives are substantially larger, implementing a big file allocation table at the beginning of the media would cause severe increases in seek time. To avoid that, `ext2fs` divides the filesystem into so-called block groups.

Each block group is equal in size, and composed of a predefined number of blocks which is calculated based on the size of a block. If compared to the FAT filesystem, a block would be similar to a cluster or a collection of sectors. `ext2fs` supports blocks ranging from 1024 to 4096 bytes, which must be defined at formatting time. How the size of a block affects the size of their groups will be explained along with bitmaps later on this section.

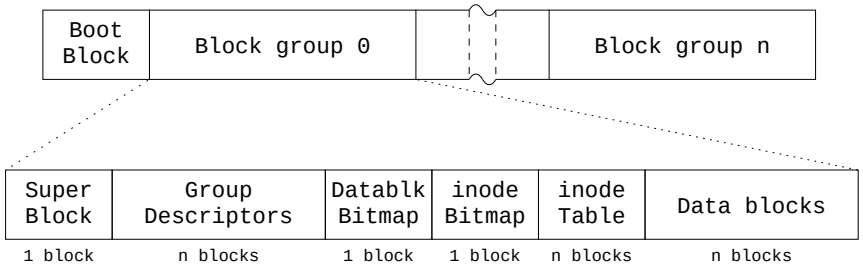


Figure 3.7.: The Extended 2 Filesystem layout.

With the concept of block explained, Figure 3.7 illustrates the structural layout of `ext2fs`. At the beginning, a block is reserved for a boot loader, similar to the reserved region that leads the FAT filesystem. Soon after that, one or more block groups follow, each one sharing an identical layout.

The first structure within a block group is always a copy of the filesystem's Super Block, which occupies one block. These copies should be identical across block groups, but usually only a few backup copies are kept up-to-date for performance reasons. The Super Block itself contains a list of these copies. On regular maintenance checks, all the others may be automatically updated. In the event of one of them becoming corrupt, a restoring utility (such as `e2fsck`) could refer to one of these backups. The Super Block is a structure similar to the BS and BPB structures of FAT, as its purpose is to hold general filesystem metadata, such as the block size and the number of free blocks.

Following the Super Block comes an array of group descriptors, which occupy 32 bytes each. The array holds copies of the group descriptor for each group and is also duplicated in each block group in the same way as the Super Blocks are for recovery purposes. A group descriptor will hold information on the position of the block and inode bitmaps, some indices and other data that help the block allocator to promptly identify the amount of free space for inodes or directories within the group.

The inode and block bitmaps occupy one block each. They are basically an array of bits, and each bit indicates if the inode or block is free (if the bit is *zero*) or not (if the bit is *one*). In view of that, supposing a block size of 1024 bytes, one block bitmap is capable of addressing availability information for 8192 blocks. That implies that each block group in the filesystem will have exactly 8192 blocks, except for the last one which might have less if the total filesystem size is not a multiple of the block group size. In the same way, no more than 8192 inodes are allowed to be addressed by the inode table, which may also occupy more than one block. This table is formed by an array of inode records for the block group, each one occupying 128 bytes. The record contains information such as the entry name, type (directory, regular file, etc.), flags, access rights and pointers to datablocks, to name a few. Differently from the concept where a file allocation table is used to map datablocks throughout the filesystem, an inode keeps track of all allocation data itself.

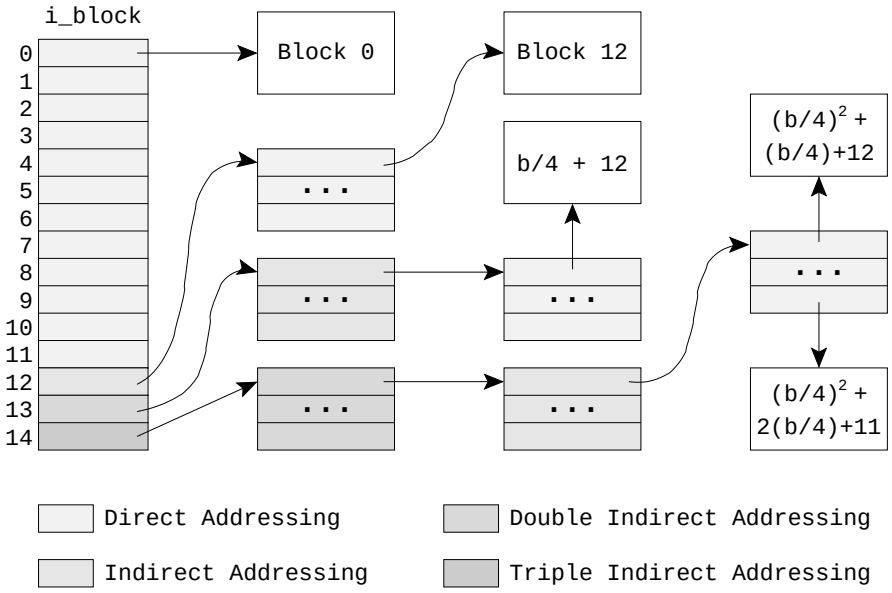


Figure 3.8.: Datablock addressing within the inode.

Figure 3.8 illustrates the `i_block` field within an inode record, which is an array of pointers to logical datablocks. By default, this array is fifteen 4 byte words long, as in the example contained in the figure, but this size may vary on custom setups. The array is divided into four different indexing schemes:

- Direct Addressing: the pointer directly addresses the logical datablock. This is used for entries of up to $12 * b$ bytes, where b is the block size, once contents can be accessed without further index seeking. The drawback is that indexing too many logical datablocks would result in bigger inode records with unused space for small entries. When the contents exceed $12 * b$ bytes, indirect addressing is used.

- Simple Indirect Addressing: in this method, a pointer will refer to a datablock which contains an array of direct addressing pointers. This array is used in the same way as explained in the previous item for the first twelve positions of `i_block`. Simple indirect addressing supports up to $b/4$ pointers per block, but requires extra reads from the storage media in order to determine the final logical datablock address. In the figure, the first block addressed would be block 12.

- Double Indirect Addressing: when the contents of a filesystem entry is larger than $(b * 12) + (b * (b/4))$ bytes, `ext2fs` starts using a double indirect addressing scheme. With this method, the logical block pointed to by `i_block[13]` will host an array of simple indirect address pointers, which will refer to the actual datablocks as explained in the previous items.

- Triple Indirect Addressing: finally, the last position of the `i_block` entry will point to a logical block that hosts an array of double indirect address pointers. The first actual datablock pointed to by this record will have logical address $((b/4)^2) + ((b/4) + 12)$, while the last will have logical address $((b/4)^2) + (2 * (b/4) + 11)$.

It is important to notice how this addressing scheme favours small files. While using indirect addressing allows for larger contents, small files (of up to $12 * b$ bytes) will always be addressed via a direct method. In this way, no extra reads have to be done to the storage media in order to obtain the subsequent indexing tables. Table 3.1 displays the amount of data that can be addressed with each method depending on the block size.

Block size	Direct	1-Indirect	2-Indirect	3-Indirect
1024 B	12 KB	268 KB	64.26 MB	16.06 GB
2048 B	24 KB	1.02 MB	513.02 MB	256.5 GB
4096 B	48 KB	4.04 MB	4 GB	$\sim$ 4 TB

Table 3.1.: Data upper limits for addressing methods.

According to these calculations, filesystems formatted with 1 KB blocks could not support files larger than about 16 GB before they would run out of pointers. The table also shows that in a 4 KB block filesystem, files up to about 4 MB could have their first 48 KB accessed by direct addressing, while the remainder would be referred to by the first indirect block. Other modern filesystems make use of *extents*, which resemble this technique but allow for files of virtually any size, since the array of pointers is organised as a tree and may contain many indirect nodes for addressing [67, 94, 44]. The real limit for file size is actually imposed by the `i_blocks` inode record, which is 32 bits long and indicates the number of 512 byte sectors addressed by the file (limiting the file size to 2 TB in `ext2fs`).

While this concludes a basic structural understanding of a block group within an `ext2fs` and therefore the structure of the whole filesystem itself, other relevant features are implemented in the kernel side of the supporting software to improve performance. Among these features, we highlight the memory structures and the block allocator, considering some of the data structures are kept in memory with a different organisation.

Because `ext2fs` has a complex structure and is not as straightforward as a FAT filesystem, the Linux kernel implements several strategies to improve overall I/O system performance. The first of these is a memory cache of the most used filesystem data structures. Table 3.2 shows the caching policy in the Linux 2.6 kernel regarding these structures, as well as the name of the representing structure on disk and memory.

Type	Disk data structure	Memory data structure	Caching policy
Super Block	`ext2_super_block`	`ext2_sb_info`	Always cached
Group descriptor	`ext2_group_desc`	`ext2_group_desc`	Always cached
Block bitmap	Bit array in block	Bit array in buffer	Dynamic
inode bitmap	Bit array in block	Bit array in buffer	Dynamic
inode	`ext2_inode`	`ext2_inode_info`	Dynamic
Datablock	Array of bytes	Buffer	Dynamic
Free inode	`ext2_inode`	None	Never
Free block	Array of bytes	None	Never

Table 3.2.: Linux 2.6 caching policy for `ext2fs` data structures.

The table shows that the most accessed data structures are always cached, namely the Super Block and the group descriptors, as most filesystem operations are likely to change them. They are loaded into the kernel page cache once the filesystem is mounted and kept there until unmounting time. On the other hand, free blocks and free inodes are never cached since they do not represent any meaningful information. In between lie structures that are loaded into memory only when they are used. For instance, an inode is kept cached while a file is open and in use.

Another interesting observation can be made by examining the difference between the second and third columns of the table. The Super Block and inode structures, while kept into memory, have different data structures associated with them. The discrepancies are given by a few extra cache pointers and counters which optimise the way the Linux kernel references and allocates data within the filesystem.

The second strategy for improving overall I/O system performance relates to the operation of the allocator. While the allocator is implemented through several different functions inside the kernel, it can be semantically divided into two distinct parts. The first part is responsible for allocating new inodes and the second part is responsible for allocating new datablocks and is invoked whenever new storage areas are required by the inode.

When allocating a new inode, the kernel first identifies whether the new entry is a directory or not. For new directories whose parent is the filesystem's root, a free inode is searched in a different block group than the parent, since it is likely that child and parent are not related. If the parent is not the filesystem's root, then the allocator will try to find an inode in the same block group as the parent. Exceptions will happen when the number of allocated inodes in the chosen block group is greater than a certain threshold related to the filesystem's average, as the kernel also tries to distribute data throughout the storage media. In those cases, neighbouring block groups would be considered better candidates.

In the case where the new inode does not refer to a directory, the allocator will always try to find space for the new entry in the same block group as the parent directory. If that group's inode table has no available entries or does not satisfy similar threshold conditions, as mentioned before, a new block group is searched using a heuristic that helps similar files for that parent directory to be allocated in similar block groups [32]. If this search also fails, a linear lookup is performed across the whole filesystem.

By attempting to keep related filesystem entries close to each other inside the media, the kernel reduces the seek time performed by the disk drive whenever these entries are being accessed. If compared to the FAT filesystem, where the file allocation table is always stored in the beginning of the partition, block groups are also an improvement since the inode tables are kept close to the actual datablocks.

Regarding the datablock allocator, whenever a new datablock is required for a filesystem entry, the Linux kernel will first try to allocate the new space contiguously to previously allocated blocks for the given inode. When that is not possible, it will try to find space inside the same block group and then move forward to neighbouring block groups and so on. By trying to allocate contiguous space prior to any other position in the partition, the allocator also helps to reduce file fragmentation.

In a last effort to reduce fragmentation, the block allocator also implements a pre-reservation technique. When invoked to allocate one datablock, up to eight other contiguous blocks are reserved for the inode. The reserved blocks are freed when the inode is closed, truncated or written in a non-sequential way, if the write operation is the one that initiated the allocation in the first place. This technique proves to be effective because when copying a file, for instance, the allocator has no way of knowing in advance the total size of the entry.

Despite the fact that these techniques make the Linux Extended Filesystems a robust solution and the default choice for most Linux distributions, they were conceived to work on a regular hard disk drive. That means that the way data is allocated and distributed across a partition may not be the best choice for a scenario where the media is composed of multiple tiers. The next chapter will discuss our experiments and proposals for a new filesystem which takes advantage of a storage media composed of physical layers with different characteristics.

3.5. The Extended 4 Filesystem

Because `ext3fs` is hard limited by design to 16 TB per filesystem, patches were written in 2006 to slightly alter the data layout and circumvent this limit. This broke compatibility of patched setups with stock kernels and therefore encouraged developers to create a new filesystem branch, preventing different installations of `ext3fs` from being incompatible. Unless noted otherwise, this section was written based on [67, 94].

Named `ext4fs`, this new branch kept many similarities with its predecessor. Like `ext3fs`, it is organised as a Super Block followed by block groups. The layout of a block group is also similar to `ext3fs`, containing the same structures in the same order (as previously shown in Figure 3.7).

Considering the developers were no longer bound by the concepts that defined `ext3fs`, the opportunity to review some of this filesystem's limitations appeared. In order to increase filesystem capacity, the amount of bits used to address the total number of datablocks was increased from 32 to 48. With 4 KB datablocks, this means filesystems could be as big as 1 EB (1024 PB) in practice. If such a limit proves to be small in the future, the developers plan to use the full 64 bits.

This led to the first difference in filesystem structure. Because every block group must keep a copy of every block group descriptor, the filesystem's size was still limited to the amount of block groups that could appear (a total of 256 TB). To fix that, `ext4fs` supports *logical* or *flexible* block groups that increase the size of datablock bitmaps and inode tables, allowing for several block groups to be considered as one. This feature also helps speeding up filesystem checks, bypassing block group size limits and plays an important role in the data allocators.

As discussed in Section 3.4, `ext3fs` allocators are heavily based on the notion of a block group, trying to keep what it considers to be similar files and directories close together in the filesystem. In `ext4fs`, because several block groups may be addressed as one, larger amounts of data can be allocated with fewer search operations for suitable space.

With the support for larger images also came the need to support larger files. `ext3fs` supports files of up to 2 TB and uses an indirect addressing scheme that is efficient for sparse or small files, but inefficient for large files [17]. `ext4fs` uses the same scheme, but if mounted with the `extent` option, can make use of *extents*, a technique already popular in other modern filesystems such as XFS [88] or JFS [44].

Extents map a range of datablocks, allowing for an inode to address a starting datablock and a length of blocks that follow. This makes even more sense with the flexible block feature, since an amount of datablocks larger than the block group can be contiguously addressed. Figure 3.9 illustrates this idea, with an extent mapping a datablock that starts in block group $\texttt{BG}_i$ and has a block count that exceeds that block group, finishing in $\texttt{BG}_{i+1}$. The shaded areas represent addressed blocks.

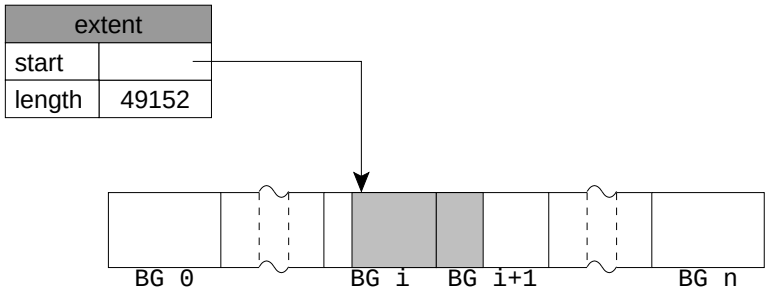


Figure 3.9.: The concept of an extent, mapping a starting datablock in a block group and having a block count that exceeds that group.

The inode structure is capable of mapping up to four extents directly in the `i_block` record. This record can be seen as a C programming language union. When mounted without the `extent` option, it behaves like `ext3fs` (explained previously in Figure 3.8). When this option is present, however, the array is treated as shown in Figure 3.10.

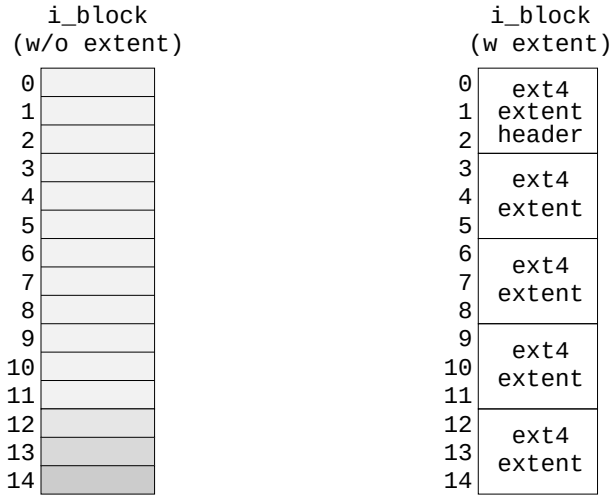


Figure 3.10.: The Extended 4 Filesystem `i_block` inode record, with and without the extent mount option.

Two different structures appear in the new `i_block` array. The first is the `ext4fs` extent header which is shown in Table 3.3 and describes how the extents are laid out for that inode. If four or fewer extents are present, then they are directly addressed by four structures present in bytes 3 to 14. A magic number is also present, allowing for new features to be added (such as a tree checksum).

```
struct ext4_extent_header
{
        __le16  eh_magic;       /* support different formats */
        __le16  eh_entries;     /* number of valid entries */
        __le16  eh_max;         /* capacity of store in entries */
        __le16  eh_depth;       /* tree depth, if any */
        __le32  eh_generation;  /* generation of the tree */
};
```

Table 3.3.: The `ext4fs` extent header in Linux kernel 2.6.20-3.

The second structure is the `ext4fs` extent which is shown in Table 3.4 and refers to datablocks allocated on the disk. It is the actual implementation of the illustrated example in Figure 3.9. The structure addresses the first logical datablock and the number of blocks covered from that point. The last 48 bits address the physical location of the first datablock.

```
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;       /* low 32 bits of physical block */
};
```

Table 3.4.: The `ext4fs` extent in Linux kernel 2.6.20-3.

When the four extents covered by the `ext4fs` inode are not sufficient to address all allocated datablocks, possibly due to external fragmentation, an extent tree is created. Every level of the tree except the bottom consists of index nodes, which are described in Table 3.5.

```
struct ext4_extent_idx {
        __le32  ei_block;       /* idx covers logical blks from 'block' */
        __le32  ei_leaf;        /* pointer to the physical block of the
                                 * next level, however leaf or index */
        __le16  ei_leaf_hi;     /* high 16 bits of physical block */
        __u16   ei_unused;
};
```

Table 3.5.: The `ext4fs` extent index in Linux kernel 2.6.20-3.

Since the extent header indicates the tree depth from that point onwards, it is possible to compute if the next level will be composed by indices or leaves. Figure 3.11 illustrates a complete example of a tree. The leftmost structure indicates the `i_block` array in the inode. Next, it is possible to observe index nodes that include extent indices pointing to leaves. Continuing to the right, the leaf nodes include extents that point to the datablocks allocated on the disk. We also note that the `ext4fs` allocation algorithms will keep the tree balanced whenever new nodes are added or removed.

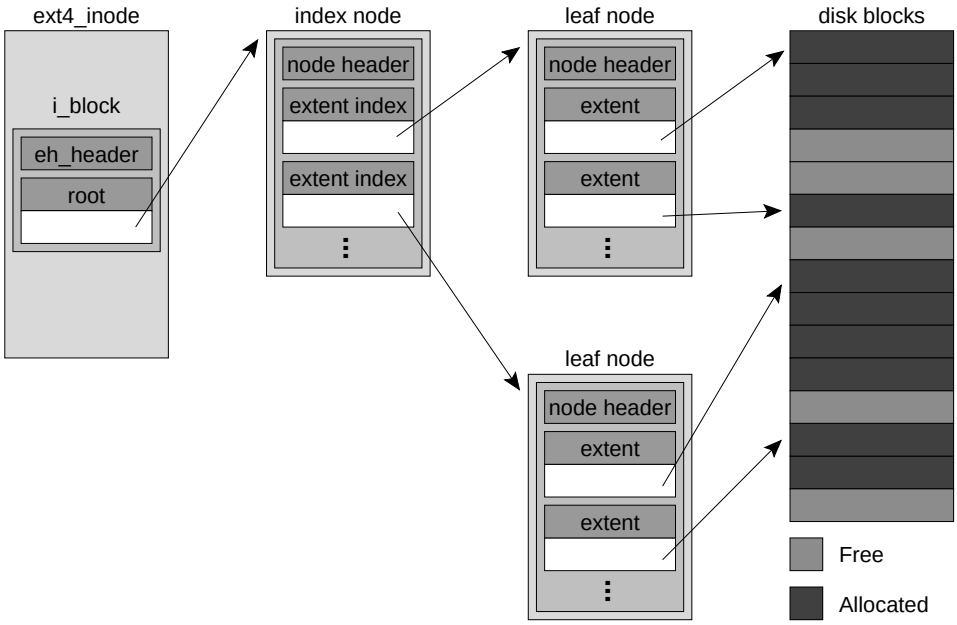


Figure 3.11.: The Extended 4 Filesystem extent tree.

Based on this new disk structure that allows larger filesystems and files while improving performance, `ext4fs` also tries to improve on other aspects. In order to minimise file fragmentation, this filesystem implements a concept called *delayed allocation*. This is a method for keeping buffers in memory for as long as possible before committing data back to the disk.

Delayed allocation helps to reduce fragmentation especially on large files by waiting a period of time before mapping memory buffers to disk. In this way, when copying a file, for example, there is time for several blocks to be allocated in memory for the destination inode, providing hints to the kernel about the amount of space that will be necessary. The allocator is also capable of deducing that it is handling a large file, and therefore should reserve more blocks than it normally would.

This is also convenient with the extent mapping scheme. If compared to the method used in `ext3fs`, where each block is addressed (in)directly, extents provide easier means for the allocation of multiple blocks to be done in a single request. Considering all requests in the Linux kernel go through a layer known as the Virtual Filesystem (VFS), which is a common interface that allows different filesystems to be implemented easily, there is no way for the `ext3fs` allocator to predict the requests that will follow.

Regarding smaller files, the allocator uses a different approach than that proposed in `ext3fs`, where files are considered similar when they are in the same directory (apart from the `root` directory, when they are considered in the opposite way). Using delayed allocation, `ext4fs` groups requests for small allocations from processes based on a per-CPU locality.

Whether this mechanism is efficient in real world situations remains to be seen. At this moment it is unclear if the small configuration files in `/etc/`, for example, that are supposed to be close together in the media for faster reading during boot time, would indeed be allocated in that fashion if created by distinct applications that are installed at different times. Studies, however, show that executing two concurrent threads, where one creates several small files and another creates one large file, will reduce fragmentation compared to similar benchmarking done with the `ext3fs` allocators [94].

Despite the efforts made in the new addressing schemes and allocation strategies, every filesystem may suffer from fragmentation especially under low space conditions. It was shown that a fragmented filesystem suffers performance penalties, especially on magnetic media [30]. An important feature proposed in `ext4fs` is *online defragmentation*, that allows for a mounted, production filesystem to have datablocks moved in order to reduce noncontiguous allocations [83].

The proposed defragmentation mechanism is usually invoked by user space processes through `ioctl()` system calls. For single files, it will attempt to reduce the number of extents by replacing noncontiguous allocated datablocks with contiguous ones. On the other hand, it may also move entire related files so they are closer together on the disk (using information such as shared parent directory). Finally, the `ioctl()` interface may also be issued for an entire filesystem.

While different strategies are applied depending on the type of the call, all algorithms are based on the idea of allocating a new temporary inode and attempting to get the required amount of datablocks as contiguous a fashion as possible. Upon success, data is transferred to the new datablocks and the extents are updated accordingly. Figure 3.12 illustrates this idea.

Concluding, `ext4fs` promises enhancements to `ext3fs`, allowing the support for larger installations and files, new techniques to boost performance and other minor features not listed in this section (such as timestamps measured in nanoseconds and larger number of subdirectories supported

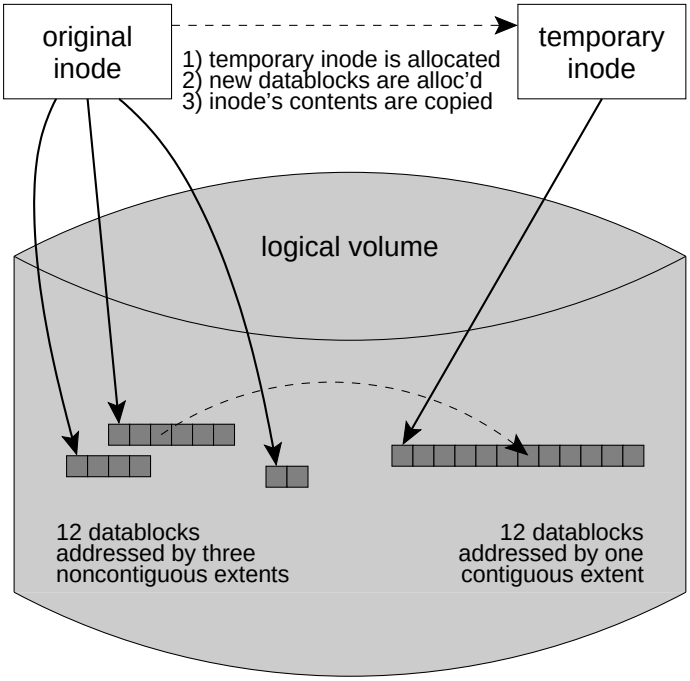


Figure 3.12.: The `ext4fs` online defragmentation strategy.

per directory, to name but a few). However, it was only marked as stable in late 2008 [63], after the start of this research, still has a limited user base and continues to receive many fixes in its implementation, subject even to structural changes. Considering these factors, we have chosen the `ext3fs` filesytem for the enhancements proposed in this research.

Nevertheless we have been able to draw inspiration from mechanisms in `ext4fs`. For example, in our implementation of the datablock migration scheme we use `ioctl()` system calls, adopting a similar principle to the online defragmentation discussed earlier in this section.

3.6. Other Filesystems

Prior to defining the filesystems we have analysed in this chapter, namely FAT and the Linux Extended 3 and 4 Filesystems, we have considered and discarded other technologies. However they were discarded for being too young, under development or simply inapplicable to the work presented in this thesis, this section will discuss some of them in brevity.

3.6.1. Btrfs

Btrfs [1], which stands for B-Tree Filesystem, was originally designed by Oracle Corporation in 2007. It is a GPL-licensed open source project that is open to contributions from independent developers. Its principal author, Chris Mason, states that Btrfs is intended to address issues such as the lack of pooling, snapshots, checksums and integral multi-device spanning in Linux filesystems, allowing for this operating system to scale upward into larger storage configurations common in the enterprise environment [33].

As the name suggests, the filesystem is based on B-Trees [12]. The original author, however, followed recommendations [79] on how to adapt such a structure for on-disk storage. This proposed modified data structure is particularly interesting for enabling Copy-on-Write techniques, and it was used in such a way that all data types are considered tree nodes.

Finally, Mason believes Btrfs will assist Linux in scaling to storage available in large enterprises not only by means of addressing large volumes, but also because Btrfs allows for administration and management via a clean interface [47]. The filesystem, however, is still under heavy development and lacks essential features such as production-ready consistency check and recovery tool.

3.6.2. ZFS

ZFS [98], which stands for Zettabyte Filesystem, was initially developed by Sun Microsystems. It is both a filesystem and a volume manager in such a way that it can be installed over multiple disks. This, combined with features such as data checksumming, allows for interesting features such as the detection of data corruption on small reads at the volume manager level (i.e. before the corrupted data is passed to the application).

After Sun's acquisition by Oracle in 2010, ZFS became part of the OpenSolaris operating system. It is licensed under the Common Development and Distribution License (CDDL) and because of that it is only available to several other operating systems as user space implementations, due to conflicts with other licenses such as GNU's GPL. A Linux native release of ZFS is still in its early stages of development, but it cannot be distributed with the Linux operating system for the aforementioned licensing reasons.

4. The Extended 3 iPODS Filesystem

This chapter describes the main contribution of this thesis: an intelligent filesystem that uses QoS hints to allocate and maintain data on multi-tier Virtualised Storage Systems. Named after the project that sponsored this research[1], the Extended 3 iPODS Filesystem (`ext3ipods`) is an enhancement of the Linux Extended 3 Filesystem (`ext3fs`). Capable of understanding the distinct QoS attributes delivered by different layers of the storage infrastructure, it uses QoS hints on datasets to intelligently allocate and maintain data layouts.

As discussed in Chapter 2, existing QoS optimisation for such storage infrastructures mostly concern boosting overall average system performance. Such solutions are based on access pattern analysis, observing the system workload including I/O requests frequencies and sizes [93, 29]. Upon inferring the access pattern of particular datasets, such solutions design new data layout configurations that may enjoy better performance. While some solutions will go as far as considering the migration cost from one scenario to another [8, 95], the design process itself has been proven expensive due to the state explosion of possible configurations [10]. Other centralised storage solutions do not consider automated strategies for such optimisations [71, 72].

Our proposal approaches the problem from a different angle. Based on the fact that certain aspects of QoS cannot correctly be inferred automatically (e.g. infrequently accessed data that require high performance), we envision a model where users and applications specify what level of different QoS attributes they would like their datasets to receive from a storage infrastructure. Such specification would happen in a model similar to the one presented in Table 4.1. The table exemplifies different datasets with

[1] Intelligent Performance Optimisation of Virtualised Data Storage Systems (iPODS), EPSRC Grant EP/F010192/1. See Section 1.4.

different storage QoS requirements, such as Log Archives that must not be stored on unreliable media, require abundant space and do not require high performance.

Datasets	QoS Requirements		
	Performance	Reliability	Space Efficiency
Temporary Data	HIGH	LOW	MEDIUM
Log Archives	LOW	HIGH	HIGH
RDBMS Datafiles	HIGH	HIGH	HIGH
Config. Files	MEDIUM	MEDIUM	LOW

Table 4.1.: Example of storage QoS requirements for different datasets.

Based on such a specification, an intelligent middleware layer should be capable of evaluating how these requirements can be best matched given a set of storage tiers, each of which provides a different level of QoS support. An ideal place to implement this middleware layer is an operating system's filesystem, since it holds metadata on all datasets and has awareness of the entire logical address range available for data allocation. In order to present a robust system capable of proving these concepts in a working environment, some key aspects needed to be considered:

- Compatibility: the new filesystem had to be compatible with existing technologies for several reasons. Firstly, to show its applicability in a realistic context. Secondly, to provide a convincing basis for comparison. Finally, to lower barriers to possible adoption. Because `ext3fs` is offered as default on most Linux distributions, we have decided to base our proposal on top of it. This has been done in such a way that regular `ext3fs` filesystems can be used as `ext3ipods` filesystems and vice versa without any conversion process between them.

- Quality of Service support: another important question is what Quality of Service (QoS) attributes should be possible to specify, at what level of detail and how these attributes should relate to actual data in the filesystem. Furthermore, what should be the means for their management, which should be realised through familiar interfaces to system administrators, users and applications.

- Data granularity: an important technical aspect for the realisation of such a filesystem is the granularity of data which will be considered

for QoS evaluation. Because QoS attributes qualify as metadata, they must relate to all disk data to which they apply. Defining the proper granularity therefore not only affects the overall filesystem performance but also causes disk space overhead.

The next sections address these matters while presenting our design for the `ext3ipods` filesystem.

4.1. Structure and Compatibility

As noted in [48], maintaining a compatible interface among different filesystems is crucial for operating systems. With `ext3ipods`, it is necessary to push that concept further. Not only must it conform to the Virtual Filesystem (VFS) layer, which is a generic interface for filesystems in the Linux kernel, but it must also be compatible with stock `ext2fs` and `ext3fs`. In order to achieve this, the `ext3ipods` filesystem must share the exact same disk structure, not only at a conceptual level (organisation through a super block, block groups, etc.), but also in every record within such structures. This allows an `ext3ipods` filesystem to be remounted at any time as a stock `ext3fs` or vice versa, avoiding any conversion process.

Considering that the proposed improvements involving QoS awareness schemes would be implemented in the filesystem components that run within the operating system's kernel (e.g. the data allocation algorithms and the attribute management interfaces), this would not be an issue. However, the metadata denoting which addresses of the filesystem provides certain QoS attributes still require persistence and therefore need to be stored inside the filesystem. Such persistence also applies to the metadata that specify what QoS attributes are required by each dataset.

This matter immediately raises the granularity issue regarding how much metadata must be saved for QoS attributes, since defining how to store them is strictly related to the volume of data which must be kept. The next two subsections address this subject.

4.1.1. Granularity of Quality of Service Metadata for Datasets

Regarding what level of granularity should be used to address datasets, the following options were considered:

- Datablock: this is the smallest reasonable granularity scheme, given that a datablock is the filesystem's unit when designating space for inode data. In `ext3fs`, its size is defined during formatting time and may vary from 1 KB to 4 KB (as previously described in Table 3.1). Implementing QoS attributes at this level could be done by declaring a structure defining the QoS metadata and instantiating a map of such structures for each block group. This map could be allocated as the contents of a special inode (such as a special file in the root of the filesystem) and allocated in reserved datablocks in each block group.

- inode: setting the granularity at the inode level produces coarser grains, since a single QoS setting refers to a group of datablocks. Implementing QoS attributes at this level could be done in two different ways. The first one is to use the inode extended attributes filesystem entry, which is a native resource to allow more metadata to be attached to any given inode. The second method is to use spare or reserved space in the inode structure, given that the QoS metadata is small enough and fixed in size.

- Block group: setting a block group as the grain size produces even coarser grains, since a single QoS setting refers to a group of inodes. Effectively, it refers to every inode allocated in that block group. However, the datablocks allocated for these inodes could be placed virtually anywhere in the filesystem. This means that changing the QoS requirement for a block group would likely result in the migration of every datablock associated with inodes in that block group.

Using a level of detail such as datablocks could be interesting for applications such as DBMS datafiles, where some records of a single file might have a different access profile than others. This would be the case for news feeds, for example, where older entries are not accessed as often as new ones. Creating an interface for such QoS specification, however, would break our

requirement for compatibility. Not only would the applications need to be modified in order to support this interface, but they would also need to be aware of the filesystem's block size, a piece of information that is usually abstracted at application level. Another issue with this level of granularity would be the severe fragmentation that would be caused when different datablocks of a single inode were assigned to different tiers within a VSS.

On the other hand, setting the granularity level to a block group would likewise present problems. Firstly, as would happen with the granularity set at the datablock level, it would be complicated to define an interface for users and applications to specify QoS attributes, considering that they are not aware of the logical addresses of allocated data. Secondly, changing the QoS attributes for a particular block group would affect every inode allocated in that block group, therefore reducing the degree of control by users and applications.

With the granularity level set to an inode, every datablock addressed by one inode is subjected to the same QoS requirements. While this grants less control than a single datablock granularity, no new interface has to be defined as there is already a set of tools which allows for the change of inode attributes. Another benefit could be reduced external fragmentation, since the default allocator attempts to place datablocks for an inode in the same VSS tier.

Regarding the storage for QoS metadata, we have looked into an inode record to find that the 4 byte attribute used to store an entry's flags has 15 bits currently not in use. Using the same interface defined for altering and displaying inode flags, those 15 bits could be arbitrarily defined for QoS definition without modifying the filesystem structure. When more space is required for QoS metadata, the inode extended attributes feature can be used, as will be discussed further in this section.

After analysing the evaluated options, the inode was chosen to represent the QoS required by datasets. This means that directories or files can carry metadata defining what QoS attributes are desired for their designated datablocks. Furthermore, means for controlling this metadata are achieved by extending existing technologies such as the toolset that is available for maintaining inode flags.

4.1.2. Granularity of Quality of Service Metadata for Infrastructure

Mapping the level of QoS that is delivered by different parts of a multi-tier storage infrastructure is a complex task due to the variations that can occur within each tier. As previously discussed in Section 2.2, the usage of technologies such as RAID combined with multi-zone hard disk drives will create scenarios where certain logical addresses will support larger throughput than others within the same tier (in the aforementioned examples, one range of the logical addresses of a tier was over five times faster than the other).

When evaluating which granularity should be used to profile the QoS delivered by the storage infrastructure, the following have been considered:

- Datablock: considering that all access at the filesystem level is done in units of datablocks, this would be the smallest amount of space to profile in terms of QoS delivered. This means that a set of QoS attributes must be defined for every datablock in the filesystem, occupied or not, incurring in a large amount of metadata overhead. Similarly to how was proposed for the granularity of QoS metadata for datasets in Section 4.1.1, a way of implementing this could be achieved by the creation of a special file in the root directory of the filesystem. Its contents could map the metadata for each addressable datablock.

- Set of datablocks: another idea would be to profile the QoS delivered by a set of datablocks, instead of doing so for datablocks individually. This would obviously reduce the amount of space required for metadata, as a single QoS attribute description would apply to a larger fraction of the storage infrastructure. Also, because datablocks sharing similar QoS characteristics are likely to be close together within a tier, the set could be classified as a range of consecutive datablocks, facilitating the grouping. Conveniently enough, `ext3fs` is already arranged as block groups, which are sets (or ranges) of datablocks as described in Section 3.4. Considering every block group is already associated with a block group descriptor, which is mainly metadata that holds information on the respective block group, this structure could be used to hold QoS metadata as well.

- Tier: one way of arranging sets of datablocks would be to group them according to the boundaries of the infrastructure tiers. Considering the examples used in this work, a tier would consist of a similar set of disks, organised in RAID arrays. As shown in Section 2.1, the main problem with this concept would be the variations within a tier regarding the QoS delivered. Using an entire tier as the granularity for QoS specification would cause inaccurate placement of data, given that all variation of the QoS delivered within that tier would not be considered.

Analysing the block group descriptor of `ext3fs` as it is implemented in Linux kernel 2.6.20-3 (the default kernel for Ubuntu 7.04 distribution, which was used during this work), we note that 14 bytes of space are not in use. Two of these bytes are padding space and another twelve are reserved for future features, as can be seen in Table 4.2.

```
struct ext3_group_desc
{
        __le32  bg_block_bitmap;        /* Blocks bitmap block */
        __le32  bg_inode_bitmap;        /* Inodes bitmap block */
        __le32  bg_inode_table;         /* Inodes table block */
        __le16  bg_free_blocks_count;   /* Free blocks count */
        __le16  bg_free_inodes_count;   /* Free inodes count */
        __le16  bg_used_dirs_count;     /* Directories count */
        __u16   bg_pad;                 /* Padding space */
        __le32  bg_reserved[3];         /* For future features */
};
```

Table 4.2.: The `ext3fs` block group descriptor in Linux kernel 2.6.20-3.

Depending on the level of profiling that is desired and the number of QoS attributes that are expected to be mapped, this space could suffice. If more metadata is needed, however, this space could be used to address one or more datablocks that would hold the additional metadata.

While these techniques solve the issue regarding the metadata storage, it is also important to evaluate if the number of datablocks contained in a block group is suitable for profiling. The actual size of a block group is defined by its datablock bitmap, therefore being proportional to the datablock size, and is defined at formatting time according to Table 4.3.

Datablock size	Number of datablocks in a block group	Block group size
1 024 B	8 192	8 MB
2 048 B	16 384	32 MB
4 096 B	32 768	128 MB

Table 4.3.: Block group sizes in `ext3fs` as defined by the datablock size.

The size of a block group can be easily calculated by considering the datablock bitmap. This is a special data block stored within the block group metadata that indicates which blocks are in use within that group. Because every bit of this bitmap corresponds to one datablock within the block group, its size is calculated as $(8 \times (datablock\_size)^2)$.

Based on the analysis made in Section 2.3, 8 MB of data (smallest size for a block group on `ext3fs`) should be a small enough grain even for a small filesystem. Considering the single Seagate ST3500630NS 500 GB multi-zoned disk analysed in Figure 2.7, the smallest zone (at the end of the disk's addressable space) is about 15 GB. Therefore even when using 4 KB datablocks, there will be about 120 grains (15360 MB/128 MB) available for QoS mapping, which is a decent amount of data to describe a zone.

4.2. Quality of Service Attributes

The previous section presented how metadata for both desired and delivered QoS attributes could be stored within the filesystem. According to the proposed methods, Figure 4.1 presents the structure of a stock `ext3fs` filesystem, similar to the one already explained in Section 3.4, but indicating (with shades) the parts that were modified to accommodate the new metadata.

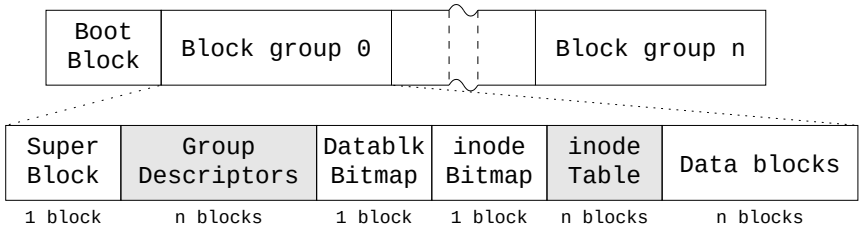


Figure 4.1.: The Extended 3 iPODS Filesystem layout.

The block group descriptors could directly store metadata for the QoS delivered by the storage infrastructure where it resides. If the metadata is too large to fit in the reserved space within the descriptor, an alternative technique could be used where indirect addressing is used and the metadata is stored in regular datablocks.

The inode table is also modified so that each inode stores metadata for the QoS required by the datablocks of that file or directory. Again, if the metadata is too large to fit within the reserved space of the stock inode structure, the inode extended attributes technique could be used. Such attributes are a resource already supported by `ext3fs`.

Having defined how QoS metadata should be stored both for requirement and delivery, it is now possible to discuss what type of attributes can be specified and how they can be evaluated. We have initially classified two different sets of attributes and will discuss them in the following two subsections.

4.2.1. Quantitative Attributes

Some attributes can be defined quantitatively. This means that a particular score can be assigned to each one of them. Similar to the examples previously used in Table 4.1 (where the performance requirements were set either to *low*, *medium* or *high*), numerical scores can be used to quantify the level of QoS. In order to allow for flexible definitions (i.e. different systems can adopt metrics of different magnitudes), our framework was defined to work with numerical scores.

In the examples used in this research, where attributes were set to *low*, *medium* or *high*, numerical representations could be 0, 50 or 100 respectively. As a matter of fact, for an environment requiring only three levels of detail, they could also be 0, 1 and 2. As will be discussed in Section 4.3, we are only interested in the difference between required and delivered levels of QoS; therefore the magnitude of the scores is irrelevant as long as it is consistent throughout the installation.

The usage of values such as 0, 1 and 2 or those of a greater magnitude such as 0, 50 and 100 depends on the needs of the environment. The latter would be preferred in storage infrastructure that are composed of tiers delivering high variation in QoS levels, in which case more than three values

are needed. Another situation that justifies the usage of higher magnitudes, with gaps between them, would be in cases where a more detailed definition of the QoS levels is expected to be required in the future.

Examples of quantitative attributes are:

- Performance: this defines how fast the I/O to a particular dataset should be performed. It could be broken down into read or write performance, or even different sized read or write operations.

- Reliability: this defines how reliable should be the infrastructure where the dataset will be stored. According to the examples previously discussed in Section 2.1, RAID1 should be more reliable than RAID5, for example.

- Space efficiency: this indicates that the dataset requires a large amount of space. It could be matched to storage media that supports compression, for example, when performance is not an issue.

- Power efficiency: while this could conflict with performance under certain conditions, it indicates that the datasets should be stored in storage infrastructures such as tape archives or MAID [22]. A MAID is an array of disks that remain spinned down unless they are being used, usually also offering some redundancy and used for infrequently accessed datasets with low performance requirements.

4.2.2. Qualitative Attributes

A different set of attributes cannot be immediately quantified by a score. It includes the set of attributes that can be better categorised in classes, indicating mainly if an attribute is required or not. Examples are:

- Enforced deletion: sensitive data need to have their datablocks wiped off the disk, sometimes more than once to erase magnetic traces, instead of simply having the inode erased. Today, a series of tools [19, 46] exists to perform recovery or forensic analysis of deleted data on the most popular filesystems, specially in cases where the datablocks were left untouched [18].

- Growth likelihood: such an attribute could help data allocators on the reservation of consecutive space for inodes in order to prevent fragmentation. Files in `/etc/`, for example, are much less likely to grow than those in `/var/log/`, therefore requiring low, if any, further space to be reserved for them.

- Regulatory compliance: some QoS requirements may apply to regulations such as SEC 17a-4 [69], defined by the U.S. Securities and Exchange Commission, that dictates the storage media on which records may be kept for broker-dealers. According to the rule, data must be preserved in non-rewriteable and non-erasable media such as Write Once Read Many (WORM) [64] storage devices.

4.2.3. Quantifying Qualitative Attributes

Ideally, the data allocator should be able to compute the difference between a numerical score for required and delivered QoS. The smaller the difference, the better the allocation scenario, considering we aim for relative QoS.

To achieve this for binary qualitative attributes, we could use scores in larger scales than those applied for quantitative counterparts. As an example, it is possible to compute the difference for performance QoS attributes in the order of the hundreds and the difference for security regulations QoS attributes in the order of the millions.

While such technique would not *enforce* that all data requiring a particular storage media would be allocated accordingly every time, it would make the selected media the preferred location whenever it is available (both present and with free space). This is in accordance to the relative nature of QoS matching which we propose and storage administrators should be aware of when selecting which attributes will be supported in their setup.

4.3. Quality of Service Evaluation

Before we consider how to enhance the inode and datablock allocators to utilise the QoS metadata, we introduce a means to evaluate a given data layout with respect to the desired and delivered QoS attributes. The fundamental idea is to calculate the difference (Δ) between the scores of QoS delivered (QoS_{dlv}) and desired (QoS_{des}), as exemplified in Equation 4.1.

$$\Delta = QoS_{dlv} - QoS_{des} \tag{4.1}$$

When the value of Δ is zero, it indicates a perfect match is in place between desired and delivered QoS for a particular attribute. This means that the closer the value is to zero, the better the data layout. When the calculated value of Δ is negative, there is underprovisioning taking place, since the QoS score desired is higher than delivered. A positive value means the data lies in a storage area that is capable of delivering a higher level of QoS; therefore overprovisioning is taking place.

In order to evaluate the layout for all the QoS attributes that are in use in a given system, an aggregate function is needed. Considering a set of attributes *attr*, Equation 4.2 presents our initial concept to realise this aggregation in the form of a sum, adding up every Δ_a calculated for each attribute a. Similarly to Equation 4.1, every Δ_a is calculated through the difference between the scores of QoS delivered (now QoS^a_{dlv}, as it is specific to attribute a) and desired (QoS^a_{des}).

$$\begin{aligned} &\textstyle\sum_{a \in \{attr\}} \Delta_a, \text{ where} \\ &\qquad \Delta_a = QoS^a_{dlv} - QoS^a_{des} \end{aligned} \tag{4.2}$$

In practice, calculating Δ_a involves traversing the inode's `i_block` array (explained in Section 3.4 and illustrated in Figure 3.8). Because datablocks for a particular inode can be allocated on any block group in the entire filesystem, it is necessary to use QoS^a_{des} for every datablock of the inode under evaluation, while using QoS^a_{dlv} for the block group where each datablock resides.

To illustrate this scenario, Figure 4.2 presents two block groups $b1$ and $b2$ that deliver QoS attributes $a1$ and $a2$ with different scores. An inode, allocated in the inode table of block group $b1$ and requiring these attributes with scores $a1 = 75$ and $a2 = 60$, has three datablocks allocated in $b1$ and another three in $b2$. The score for this example is obtained by summing the differences of QoS_{dlv} and QoS_{des} for each datablock, considering the block group where it resides. Table 4.4 shows how the calculation is done.

The scenario proposed does not contain block groups capable of delivering *exactly* the same score required by the inode. Despite that fact and according to the steps illustrated in the table, the resulting score for this data

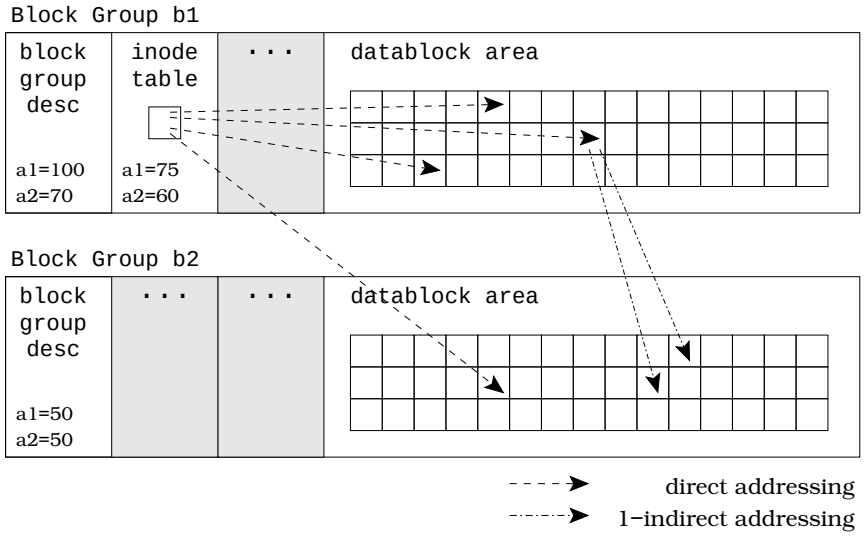


Figure 4.2.: Example, for QoS evaluation, of an inode with datablocks allocated in two different block groups.

$\sum$	$(3 \times (100 - 75))$	Three datablocks requiring score 75 for attribute $a1$, residing in block group $b1$ that delivers score 100 for the same attribute.
	$(3 \times (50 - 75))$	Three datablocks requiring score 75 for attribute $a1$, residing in block group $b2$ that delivers score 50 for the same attribute.
	$(3 \times (70 - 60))$	Three datablocks requiring score 60 for attribute $a2$, residing in block group $b1$ that delivers score 70 for the same attribute.
	$(3 \times (50 - 60))$	Three datablocks requiring score 60 for attribute $a2$, residing in block group $b2$ that delivers score 50 for the same attribute.
$=$	0	The resulting score is *zero*, although there is both over- and underprovisioning happening, suggesting that a perfect layout is in order.

Table 4.4.: Step-by-step QoS evaluation of an example containing datablocks scattered over two different block groups.

layout is *zero*. This suggests that a perfect layout is in place, with respect to QoS evaluation, even though data allocated in $b1$ is overprovisioned and data allocated in $b2$ is underprovisioned.

This problem raised the question of evaluating under- and overprovisioning cases with a different method. In order to achieve this, the QoS evaluation formula was modified to include a provisioning factor p. Equation 4.3 presents this enhancement.

$$\begin{aligned} &\textstyle\sum_{a\in\{attr\}} \Delta_a \times p, \text{ where} \\ &\Delta_a = QoS^a_{dlv} - QoS^a_{des}, \text{ and} \\ &p = \begin{cases} -|c_u| & \text{if } \Delta_a < 0 \\ |c_o| & \text{if } \Delta_a \geq 0 \end{cases} \end{aligned} \tag{4.3}$$

Using the new evaluation formula, the sum will always be composed of positive numbers, since p will switch the sign of Δ_a if it is negative. To differentiate the provisioning scheme the formula offers c_u and c_o, which are constant factors applied to indicate how worst it is to under- or overprovision. For example, if one system considers underprovisioning twice as bad as overprovisioning, c_u and c_o may be set to 2 and 1 respectively, adding twice the score every time Δ_a is less than zero.

Re-evaluating the previous example with the new formula and the proposed values for c_u and c_o, it is possible to observe how the final score is affected by data that is not perfectly matched. Along with these new results, we also present cases where all datablocks are either in block group $b1$ or $b2$. The results are expressed in Table 4.5.

Evaluated Score	**Scenario Description**
315	Same scenario as before, having three datablocks allocated in block group $b1$ and three datablocks allocated in block group $b2$.
420	Scenario with most underprovisioning, where all six datablocks have been allocated to block group $b2$.
210	Scenario with most overprovisioning, where all six datablocks have been allocated to block group $b1$.

Table 4.5.: Data layout evaluation of three different scenarios using the provisioning factor.

Comparing the results of the original evaluation scheme with the new one that takes provisioning into account, it is possible to observe how the

scores are more informative. Firstly, none of the results indicate perfect allocation, considering there are no block groups capable of delivering exactly what is desired. Secondly, the new results show that if some or all data is underprovisioned, the resulting scores are higher than if all data is overprovisioned.

Finally, when applying the score evaluation formula to calculate possible layouts with the intent to allocate data, we considered situations where ties could happen between different attributes. For example, if a scenario presents a situation where it is impossible to meet both a performance and a reliability requirement, possibly because the infrastructure is unable to deliver high values for both attributes simultaneously, breaking the tie may be required. In order to achieve this, one last multiplier factor m_a is added to the formula, as shown in Equation 4.4.

$$\begin{aligned} &\textstyle\sum_{a \in \{attr\}} \Delta_a \times p \times m_a, \text{ where} \\ &\Delta_a = QoS^a_{dlv} - QoS^a_{des}, \\ &p = \begin{cases} -|c_u| & \text{if } \Delta_a < 0 \\ |c_o| & \text{if } \Delta_a \geq 0 \end{cases}, \text{ and} \\ &m_a = \text{attribute multiplier} \end{aligned} \tag{4.4}$$

The attribute multiplier could also be used to reinforce the importance of certain qualitative QoS attributes. As previously discussed in Section 4.2.3, where it was proposed that a different scale of scores is used for such attributes (i.e. scores in the order of thousands or millions instead of hundreds), it is now possible to use m_a causing the score to behave accordingly.

4.4. Quality of Service Management Interfaces

As discussed throughout this chapter, there are two types of QoS definitions that are applied in `ext3ipods`. One is defined by users or applications over datasets and the other is defined by a storage administrator when profiling delivered QoS. In view of that, there are two distinct QoS management interfaces.

4.4.1. inode Management Interface

One of the reasons for selecting the inode flags and extended attributes support for the implementation of our QoS metadata was the existing management mechanisms. The inode flags are contained within the inode structure and are managed through `ioctl()` system calls. Extended attributes are a list of *name:value* pairs, managed through a dedicated library called `libattr` and stored in dedicated datablocks.

Considering this work focuses on a small set of QoS attributes to prove the concepts introduced by this thesis, we will work exclusively on the inode flag support. As discussed in Section 4.1.1, given the amount of QoS metadata is fixed in size and small enough to fit in reserved space of the inode structure, there is no need to use additional datablocks.

While the generic kernel interface can be used directly by applications to retrieve and alter inode flags through the `ioctl()` system call, libraries and sets of tools exist to facilitate the task. We point out the `libe2p` library, which is developed and maintained along with the `e2fsprogs` [90] suite of tools. Besides being present in all major desktop and server Linux distributions, they allow for easy integration with existing scripts and applications.

The `e2fsprogs` include two tools named `lsattr` and `chattr` for viewing and setting inode flags respectively. They are used in the same straightforward manner as `chmod` is used to manipulate the read, write and execute flags (as well as other bitsets such as setuid). To maintain elegance and compatibility, these tools make use of `libe2p` in order to access the inode's flags instead of issuing `ioctl()` calls directly.

Users managing files and directories through a shell environment can call `lsattr` and `chattr` directly. Similarly, user scripts can call these tools in the same manner. Applications may manage inode flags with a greater range of options. Depending on the scenario, they may call scripts or the `e2fsprogs` shell tools directly. The preferred method, however, would be to compile the applications against `libe2p` for improved performance, elegance and compatibility. Ultimately, they could issue `ioctl()` calls directly, although bypassing the library is not recommended considering the overhead to keep applications up-to-date when changes are made to the attribute support. Figure 4.3 illustrate this integration process, showing the possible interactions between the relevant components.

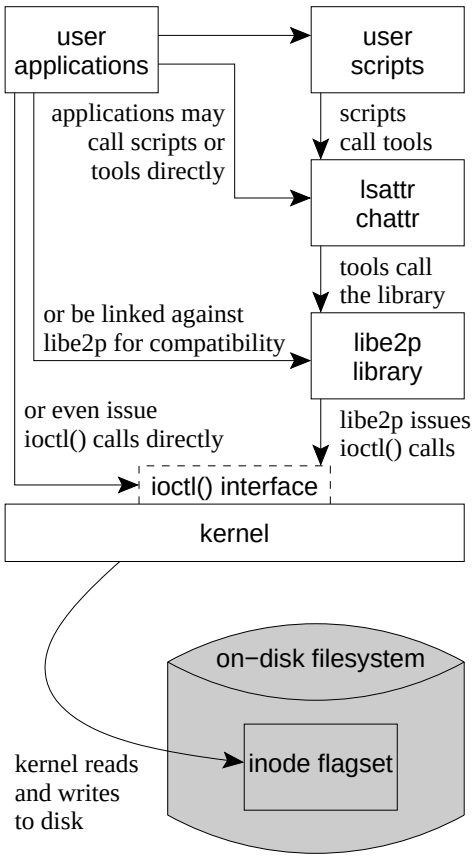


Figure 4.3.: Interface integration for inode flags manipulation.

In order to implement support for our QoS metadata, we have mapped a fixed set of attributes to use throughout this work. This set is composed of read performance, write performance and reliability. Firstly, we have modified `libe2p` to understand the new flagset which includes these attributes. Next, we have enhanced `lsattr` and `chattr` to properly parse command line options and to issue library requests accordingly. The kernel implementation of `ioctl()` system calls did not have to be modified, considering we use the stock read and write inode flags options. This reinforces the preference for application interaction via the library and not with the `ioctl()` kernel interface.

4.4.2. Block Group Management Interface

At the other end of the storage infrastructure, system administrators require an interface to manage the flags in the block group descriptors describing provided QoS. Because block group descriptors are composed of metadata that is managed solely by the kernel, there is no straightforward interface available to users, applications or system administrators capable of managing their contents.

While there are different means for user space applications to communicate with the kernel, two popular choices are device files (special files located in the `/dev/` directory) or the `proc` filesystem. Considering the `proc` filesystem does not need to be mounted at all times, we have created a device file for system administrators to interact with the block group descriptors.

Device files may be of two different types [73]. The first one is called a block device and is used for the transfer of data in sets of bytes, usually sectors, with a typical example being storage media such as hard disk drives. The second type is called character device and is used when the transfer of data is realised one byte at a time, with an example being terminals. Considering our interface must support the communication of varying sized commands and return buffers, we have used the latter.

On the kernel end, we have developed a loadable module named iPODS Filesystem Manager (`ifm`) which interacts with user space requests through the character device `/dev/ifm`. Among other features which will be discussed later, this module is used to manage QoS metadata from block group descriptors of an `ext3ipods` mounted filesystem.

Using this method for managing a mounted filesystem provides additional benefits. For instance, system administrators may profile the storage infrastructure and update QoS metadata on-the-fly without the need to disrupt the filesystem usage. Additionally, `ifm` interacts directly with the kernel cache of the set of block group descriptors, providing improved performance.

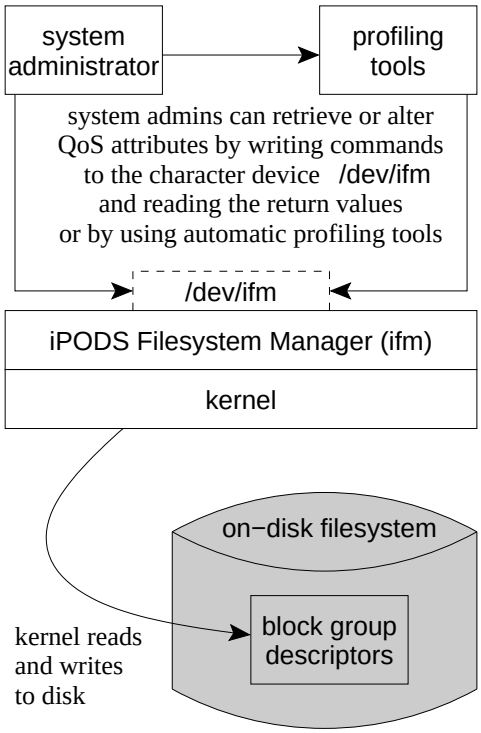


Figure 4.4.: Interface for block group descriptors metadata management.

Figure 4.4 illustrates the block group descriptors metadata management process. It is important to note that system administrators may write and read directly to the character device, using terminal commands such as `echo` and `cat` or develop automatic mechanisms. For example, automatic performance profiling tools [15] could update the relevant performance QoS attributes by holding open a file descriptor on `/dev/ifm` and reading or writing the appropriate commands.

4.5. Allocation Strategies

We now discuss three different allocation strategies that we made use of while conducting this work. Firstly, we will describe the basics of the default allocator present in `ext3fs`. Secondly, we will show how we modify the inode allocator to intelligently make use of the QoS attributes. Finally, we go further and present modifications to the datablock allocator so it also makes use of QoS attributes.

4.5.1. Default `ext3fs` Allocator

Naturally, the default `ext3fs` inode and datablock allocators are completely unaware of any QoS metadata in the filesystem. Not only the QoS metadata introduced by by `ext3ipods` uses reserved space that is originally ignored by `ext3fs`, their algorithms are exclusively focused on keeping directories and files that are alike close to each other on the physical media and on reducing fragmentation.

To achieve this, the *inode allocator* exploits the concept of block groups. When finding a place for a new inode, the allocator will attempt to use the same block group for entries that reside in the same directory. The only case where that is not true is when creating subdirectories on the filesystem *root*, in which case the allocator assume these entries do not have anything in common and therefore it will do the opposite: that is, spread them into different block groups that are far apart.

After allocating an inode, data is usually written to the media, with the corresponding datablocks being associated with the new inode. The *datablock allocator* is responsible for finding where to write. Because our focus is on allocation regarding QoS matching, our experiments will not include concurrent writing or other fragmentation-inducing experiments.

While searching where to write, the datablock allocator will try to find new space in the same block group as the inode when dealing with the first datablock to be allocated for the inode. In cases where previous allocation has been done, the kernel attempts to allocate contiguous space. When that fails, it will search for space in the same block group. This mechanism is particularly relevant to this thesis, as in some cases the use of our intelligent inode allocator alone already provides benefits in terms of data placement when considering QoS requirements.

When there is no space available within the current block group or there is an imbalance of inode modes in that block group (meaning that a certain ratio of directories/files is high), the datablock allocator will search for space elsewhere, giving preference to block groups close to the inode or to the last datablock allocated. This also concerns our new datablock allocator, that is enhanced with additional criteria as described below.

This algorithm was initially proposed by Grigoriy Orlov upon noticing that small filesystems performed better than their larger counterparts [74]. Despite the fact that it has been accepted by the community and used in `ext3fs` since November 2002 [89], little benchmarking has actually been done to prove its benefits [43].

4.5.2. QoS-aware inode Allocator

As discussed in Section 4.5.1, the original `ext3fs` inode allocator adopt different strategies for directories or files in an attempt to spread data unlikely to be related across the filesystem. Our goal is slightly different, as we aim to group data that has the same QoS requirements in the set of block groups that is capable of delivering those requirements the best. To achieve this, we completely replaced the default allocation algorithm with a new strategy.

Our strategy uses the QoS match score formula presented in Section 4.3 to scan for block groups capable of providing a score of 0 (which is a perfect match between what is required by the inode and delivered by the block group). Naturally, it only consider block groups with free space.

In case there are no block groups capable of perfectly matching the QoS required by the inode, our allocator will choose the one closest to 0. This search is currently realised linearly from the first block group of the filesystem towards the last. Algorithm 1 lists this idea in pseudocode.

Lines 1 to 5 of the algorithm declare the variables that are going to be used. $Group\_idx$ is the iterator used in the loop during the search and $nGroup$ is the total number of block groups. $BestGroup$ stores the index of the best suitable group located and is initialised to -1, indicating that none was yet found. $Score$ annotates the current score should the inode to be allocated be placed in a certain block group. It is initialised to 0 and $BestScore$, which indicates the best score computed during the search, will be initialised to the maximum value that the variable type can store.

Algorithm 1: QoS-aware block group locator for inode allocation.

```
1  Group_idx = 0;
2  nGroup = total_number_of_block_groups;
3  BestGroup = −1;
4  unsigned Score = 0;
5  unsigned BestScore = Score − 1;
6  while Group_idx < nGroup do
7  |  if GroupDsc[Group_idx].isFull then
8  |  |  Group_idx++;
9  |  |  continue;
10 |  endif
11 |  Score = EvalQoS(GroupDsc[Group_idx].Delivered, Inode.Required);
12 |  if Score = 0 then
13 |  |  BestGroup = Group_idx;
14 |  |  BestScore = Score;
15 |  |  break;
16 |  else
17 |  |  if Score < BestScore then
18 |  |  |  BestScore = Score;
19 |  |  |  BestGroup = Group_idx;
20 |  |  endif
21 |  endif
22 |  Group_idx++;
23 endw
24 if BestGroup = −1 then
25 |  return error: no space available;
26 endif
27 return: Group_idx;
```

The value of *BestScore* depends on the architecture used (32 or 64 bit) and the magnitude of the QoS levels (as discussed in Section 4.2.1). When choosing the magnitude of QoS levels that are going to be used in a certain installation, system administrators should mind their architecture storage limits to avoid variable overflows during this calculations.

Next, lines 6 to 23 present the loop that browse all block groups from 0 to *nGroup*. Firstly, lines 7 to 10 ensures there is free space in that block group, incrementing the iterator and continuing otherwise. Following that, line 11 issues a *EvalQoS*() call, which in practice is the application of Equation 4.4 introduced in Section 4.3. The input values are the QoS levels delivered as described in the block group descriptor *GroupDsc* and the QoS levels required by the new inode *Inode*.

In case a score of 0 is achieved, indicating a perfect location was found for the inode, the *if* statement in lines 12 to 16 will update the values of *BestGroup* and *BestScore*, interrupting the algorithm. Otherwise, the *else* statement in lines 16 to 21 will keep the values of *BestScore* and *BestGroup* updated with data of the best locations found so far. This is the realisation of the *relative QoS* concept, considering our idea is to provide datasets with the best possible match given their QoS requirements and the infrastructure availability.

Finally, lines 24 to 26 verify that, if *BestGroup* still is -1, there are no space available in the filesystem. This is proven by the fact that, unless the *continue* statement was executed on line 9 during every iteration, *BestGroup* must have been altered at least once either in line 13 or in line 19.

Possible Enhancements

Observing the algorithm, it is possible to notice that caching the results for *EvalQoS*() would likely improve the overall performance of the search procedure. Unless cache querying and updates take longer than calculating the match (perhaps depending on the number of QoS attributes present on a particular system), this would prevent the same calculations to be executed with the same values.

However, there are at least a couple of complications in this optimisation process. Firstly, the result of the formula is different depending on

the attributes of each block group and those of the inode. This means that the cache size would vary dramatically depending on the number of QoS attributes present in a system and the magnitude of the values used. Secondly, the attributes both in block groups and inodes may change overtime, with each of these changes affecting large portions of the cache.

Another attempt to reduce the search process would be to facilitate a 0 match to happen. In order to achieve that, we might think to use a QoS match score formula with $c_2 = 0$ (i.e. so that we consider overprovisioning as a perfect match). After experimenting with some scenarios, as will be shown in Chapter 5, this proves not to be ideal due to the rapid exhaustion of inode space in block groups capable of offering high QoS parameters.

4.5.3. QoS-aware Datablock Allocator

As already discussed, the original `ext3fs` datablock allocator attempts to reduce file fragmentation by keeping datablocks close to each other (trying to reduce seek time when reading large chunks of data). While we do not wish to create external fragmentation, our main concern is to keep required and delivered QoS as closely matched as possible. In light of this, several ideas were considered.

Initially, we evaluated the possibility of using the inode's block group as an allocation goal. This proved complicated due to the backwards compatibility of `ext3ipods`. Considering inodes could have been allocated while mounted with a stock `ext3fs` kernel, there is no guarantee that its placement was based on QoS-aware features.

However, this idea could be adopted for the allocation of the first datablock of an inode, in those cases where QoS attributes were considered in the placement of the inode, since most applications will immediately write data after allocating an inode. Unfortunately, the set of functions used by the datablock allocator does not include resources for passing such information, creating implementation complications for this idea. Additionally the allocator would not be able to distinguish between allocation requests made immediately after the creation of an inode and those made after an inode has been truncated to zero bytes, bringing us back to the original issue of the stock `ext3fs` kernel. This proposal was therefore dropped due to implementation-specific complications.

Another idea that was considered regards using the principle of the stock `ex3fs` allocator. That is, when allocating a new datablock, attempt to obtain that datablock which immediately follows the last allocated datablock for an inode. As discussed, this is originally done with the intent to prevent external fragmentation. Our intent, however, is to take advantage of the cases where the last datablock was allocated considering the QoS element. We therefore considered relaxing that original constraint to allocate the new datablock on at least the same block group as the last allocated one. However, similar complications arose as there was no guarantee that the last allocated datablock was realised based on QoS features.

While we believe that fragmentation is an important issue that must not be disregarded, even in solid state drives [20], we have limited the focus of this study to the placement of data regarding QoS matching. In the future, if experiments showing that concurrent allocation causes external fragmentation while using our algorithm, considerations must be made to weight this factor and consider contiguous allocation depending on the QoS penalties incurred.

Finally, we idealised a QoS priority queue. For every datablock allocation request, a list of block groups is elaborated and sorted according to how well the QoS attributes desired by the inode owning the request are met with respect to the QoS attributes delivered by a particular block group. Once the priority queue generation is completed, attempts are made to allocate datablocks from the best possible block group until the last. This strategy is represented in pseudocode by Algorithm 2.

Lines 1 to 3 of the algorithm declare the variables used. $Group\_idx$ is the index used to iterate through block groups. $nGroup$ stores the total number of block groups in the filesystem. Finally, $pQueue$ represents the priority queue, which is kept sorted according to the best possible QoS matches between block groups and inodes.

Following that, the loop indicated from lines 4 to 12 populates the priority queue. The initial check in lines 5 to 8 will skip block groups that are full. The $EvalQoS()$ call issued on line 9 is equal to that in Algorithm 1, and is the evaluation done by Equation 4.4 presented in Section 4.3. Next, the $Score$ is associated with the group index and inserted in $pQueue$. The insertion procedure should guarantee that the list remains sorted according to the $Score$. Finally, $Group\_Idx$ is incremented and the loop continues.

Algorithm 2: QoS-aware block group locator for inode allocation.

```
1  Group_idx = 0;
2  nGroup = total_number_of_block_groups;
3  pQueue = empty_linked_list;
4  while Group_idx < nGroup do
5  |  if GroupDsc[Group_idx].isFull then
6  |  |  Group_idx++;
7  |  |  continue;
8  |  endif
9  |  Score = EvalQoS(GroupDsc[Group_idx].Delivered, Inode.Required);
10 |  pQueue.insert(Group_idx, Score);
11 |  Group_idx++;
12 endw
13 while pQueue.hasItems do
14 |  Group_idx = pQueue.popItem();
15 |  if allocateBlock(Group_idx) = success then
16 |  |  free pQueue and return: block allocated at Group_idx;
17 |  endif
18 endw
19 free pQueue and return error: no space available;
```

With the priority queue completed, the algorithm executes another loop from lines 13 to 18. This loop will pop indexes from *pQueue* (line 14) and issue an attempt to allocate a datablock from the refereed block group (lines 15 to 17), returning upon success. Here, we assume that an allocation failure will throw an exception and end the execution. If line 19 is reached, no allocation has succeeded, and the algorithm returns with an error. Before executing any of the return statements, it is important to guarantee that the priority queue was properly deallocated to prevent memory leaks.

Possible Enhancements

Similarly to what has been proposed for the QoS-aware inode allocator, a cache containing the results for *EvalQoS*() could be created. This would prevent the function from being executed repeatedly when allocating datablocks for the same inode or for inodes with the same QoS requirements. Naturally, similar complications will occur, mainly relating to the overhead of maintaining a consistent cache considering factors such as QoS attribute changes overtime.

The next possible optimisation would be to cache the priority queue containing which block groups provide a good fit for newly allocated datablocks. Again, a trade-off would have to be evaluated in terms of the benefits for speed of allocation against the costs for maintaining such cache. The results of this allocation are likely to depend heavily on system-specific data layouts and QoS configurations.

Because the QoS evaluation of how well datablocks fit into block groups is similar to the one used for inodes, similar problems are likely to be encountered. Furthermore, the datablock allocation is broken down into separate functions within the Linux kernel implementation. Firstly, a block group is selected and only then a proper allocation attempt is made. This second step could fail, for example, due to exhaustion of space during race conditions.

Finally, depending on the tier size for certain infrastructures, it is likely that several block groups will deliver QoS attributes at the same level. This facilitates for the implementation of an extension based system, where a map could provide the QoS attribute levels for a range of block groups, similar to the addressing scheme of `ext4fs`. While the implementation considerations of such mapping needs further study, it would provide benefits for both algorithms due to a reduction in the number of iterations for the loop that locate candidate block groups.

To keep the focus of this work within the proof of the benefits of QoS-aware allocation, we leave algorithm enhancements for future work.

4.5.4. Concurrent Execution

Previously in this section, we have explained that concurrent experiments for the purposes of inducing internal file and directory fragmentation would not be conducted. This decision was made due to the nature of this research, which leaves such studies for future work.

Concurrent execution of the newly proposed algorithms, however, would be handled in the same way the the default `ext3fs` allocators are handled. Due to the reentrant and preemptive nature of the Linux kernel, the inode and datablock allocators are designed to be thread safe. That is, multiple threads executing the algorithm, even if running concurrently on multiple CPUs, will not affect each other.

Furthermore, the proposed algorithms are, in practice, helper functions that return a datablock or an inode for allocation. The kernel resource that actually allocates these targets is developed in such a way to handle the case where the indicated target is no longer available. This caters for the case where a datablock or an inode is selected for allocation by two or more concurrent executions of the algorithm.

4.6. Migration Strategies

4.6.1. Motivation

However efficient the algorithms presented in Sections 4.5.2 and 4.5.3, there are scenarios where data layouts may reflect poor allocation QoS-wise. This could occur for a number of reasons:

- Pre-existing filesystem: due to the compatibility features of `ext3ipods`, it is possible that data has been allocated using stock `ext3fs` allocators. Considering these allocators lack with regards to QoS elements, the QoS evaluation of the data layout would inevitably show poor allocation.

- Data life cycle: considering it is possible to modify inode or block group attributes after they were initially set, there might be cases where data was properly allocated, but its attributes have changed and the new QoS evaluation shows poor allocation. Studies showing data evolution in large-scale filesystems were presented in [31].

- Misallocation of data: besides pre-existing filesystem, other factors may cause data to be misallocated according to QoS requirements. This is likely to happen in filesystems working close to full capacity, when block groups capable of providing appropriate space for certain QoS requirements have no space available.

- Changes to the storage infrastructure: given that new tiers can be dynamically added to existing filesystems, it is possible that new QoS configurations are defined on block groups. This means that existing QoS requirements can be matched better to a different storage area according to our evaluation mechanisms.

Regardless of the cause for data not being located at the most suitable storage area according to our QoS evaluation mechanisms, data migration provides means to improve that scenario. It has already being shown that data migration, under the appropriate conditions, is capable of providing benefits and cost improvements [36, 84]. The next subsection discusses how to decide the right moment to evaluate such conditions.

4.6.2. Triggers

While migration is an interesting resource to improve data layout in a filesystem, it may not be obvious to realise when it becomes necessary. We have identified possible triggering mechanisms to initiate the reevaluation of an inode, a set of inodes or the entire filesystem as follows:

- Active migration: initiated by an external factor, such as system administrator requests or QoS attribute changes to existing inodes or block groups, this is the process of actively migrating data on a mounted filesystem. It could be issued for a particular set of inodes or for an entire filesystem, causing the kernel to evaluate whether the specified datablocks could have a better QoS score if placed elsewhere.

- Passive migration: initiated by the kernel, either on a regular basis or during periods of low I/O activity in the system, this is the process of attempting to improve the data layout regarding QoS evaluation on a mounted filesystem. Passive migration could also be triggered when datablocks are read, modified and flushed back to disk from the cache, in which case the kernel would evaluate the possibility of writing it back to a different location.

- Offline migration: similar to `ext4fs` offline defragmentation discussed in Section 3.5, this process would be initiated by a system administrator on a filesystem that is not mounted. Like the other scenarios, it could also be executed to a set of inodes (including a single one) or the entire filesystem.

It is important to notice that the migration of inodes should be avoided. Considering the nature of inode tables in `ext3fs`, it is possible for several directory entries to point to a single inode (a technique known as hard

linking). Furthermore, the inode index is obtained through its position in the inode table of a particular block group, meaning that migrating an inode for a different block group would cause its index to change. Because there is no straightforward manner to obtain pathnames from inode indexes [40], traversing the entire directory tree of a filesystem to update such indexes would not be feasible.

4.6.3. Migration Strategies

Algorithms for the evaluation of data layout and the proposal of migration plans have been elaborated in the literature, but exclusively considering the access patterns and frequencies relating to the performance characteristics of a storage system [95, 8, 29, 93]. Other studies involving mapping of migration plans, optimisation and cost evaluation of object placements in networks have been conducted in [6, 39].

Despite these studies having promising results, a proposal that relates better to what we are trying to achieve has been done in the `ext4fs` online defragmentation previously discussed in Section 3.5. That process strives to reduce datablock fragmentation by allocating new contiguous areas and copying data into them. In our model, we can use the same principle to move datablocks with low QoS scores.

Detecting if a QoS score is low, however, is more complicated than observing if a file is fragmented. This is because it is not possible to directly infer if a better score could be achieved in a different allocation scenario. To obtain that information, it is necessary to evaluate the desired QoS against the delivered QoS for every block group in the filesystem. This process involves traversing the directory tree of a filesystem and evaluating the QoS score for every inode.

Furthermore, every inode could contain datablocks allocated in different block groups. This means that, for every inode candidate to migration, it is necessary to test if it is possible to improve the allocation of its datablocks on an individual basis. Considering these elements, we present Algorithm 3.

Lines 1 and 2 declare the variables used in this algorithm. $DBlock\_idx$ is the loop iterator index for the datablocks and $nDBlock$ indicates the total number of datablocks allocated to the inode. Next, lines 3 to 18 list the main loop that iterates through all candidate datablocks.

Algorithm 3: QoS-aware migration algorithm.

```
1  DBlock_idx = 0;
2  nDBlock = total_number_of_datablocks_in_inode;
3  while DBlock_idx < nDBlock do
4      cGroup = getGroupDsc(DBlock_idx / total_blocks_per_group);
5      cScore = EvalQoS(cGroup.Delivered, Inode.Required);
6      if cScore > 0 then
7          tDBlock = Allocate_DBlock(Inode);
8          tGroup = getGroupDsc(tDBlock.idx / total_blocks_per_group);
9          tScore = EvalQoS(tGroup.Delivered, Inode.Required);
10         if cScore > tScore then
11             oDBlock = getBlock(DBlock_idx);
12             copy contents of oDBlock to tDBlock;
13             update inode to point to tDBlock.idx instead of DBlock_idx;
14             free oDBlock;
15         else
16             free tDBlock;
17         endif
18     endif
19     DBlock_idx++;
20 endw
21 return;
```

The loop consists of obtaining the current allocation QoS score for a datablock and, if that score is not 0, attempting to allocate a new datablock with a better score. If this succeeds, the contents of the current datablock are migrated to the newly allocated one and the pointers are updated in the inode.

Line 4 obtains the block group descriptor for the datablock at hand through an integer division of the block index by the number of blocks per group (which is filesystem constant). Line 5 computes its allocation QoS score using that information. Next, the if statement from lines 6 to 18 limits actions only when the score is greater than 0, given that it would otherwise not be possible to improve the allocation scenario.

The *Allocate_DBlock*() call in line 7 executes Algorithm 2 without binding the newly allocated datablock to the inode. Assuming that the allocation was successful, line 8 obtains the block group descriptor of this new datablock and line 9 evaluates its QoS score.

The if statement in lines 10 to 17 performs the migration process and is executed only if the newly calculated score is improved when compared to the original one for that datablock. If the score has not improved, the new datablock is released and the algorithm loops back to line 3 after incrementing *DBlock_idx* in line 19.

The migration process described in lines 11 to 14 consists of four steps. Firstly, a handler for the old datablock *oDBlock* is obtained based on *DBlock_idx*. Secondly, the contents of the old datablock are copied to the new one. Thirdly, the inode pointers for that datablock are updated. It is important to note that due to the addressing scheme in the `ext3ipods` filesystem, this could mean updating the `i_block` array in the inode structure or update pointers in other datablocks in case of indirect addressing. Finally the old datablock is released, meaning its index is freed in the original block group descriptor.

Possible Enhancements

Originally, we believed that it was not necessary to attempt the migration process for every datablock of an inode. Once it was not possible to improve the allocation of a datablock, it appeared impossible to improve the score of subsequent datablocks. Further analysis proved this was not the case.

Considering inodes have datablocks allocated at different times, the filesystem situation can have changed in terms of free space and QoS levels, independent of the time frame. This could cause previous allocations to have happened with different scores than newer ones. Apart from that, inodes could also have had QoS attributes changes overtime and further datablocks allocated to them prior to a migration trigger. Based on these reasons, we believe it is necessary to attempt the migration process for every datablock associated with an inode.

We note that, based on the enhancement ideas proposed for the allocation algorithms in Sections 4.5.2 and 4.5.3, access to the $EvalQoS()$ function both in lines 5 and 9 could be cached.

5. Experimental Results

This chapter presents several sets of experiments conducted with `ext3ipods`. Firstly, we discuss the hardware and software configurations of the environment in which our tests were executed. Secondly, we present the suite of tools that was developed to perform QoS evaluations of data layouts as well as visualisations. Thirdly, we show how QoS extensions were adapted to the benchmarking framework Impressions [3]. Finally, we evaluate several filesystems populated by Impressions to show the benefits of our solution.

5.1. Test Environment

Our first step towards assembling a practical experimental environment was to analyse the hardware at our disposal. The storage infrastructure used was an Infortrend EonStor A16F-G2430 enclosure connected via a dual fibre channel interface to a Ubuntu 7.04 (kernel 2.6.20.3) Linux server with two dual-core AMD Opteron processors and 4 GB RAM.

To facilitate the experiments, we considered using the same tiers previously evaluated in Section 2.3; that is, a 1.5 TB RAID5, a 1 TB RAID10 and a 2 TB RAID0. These tiers were concatenated as described in Section 3.1 using Linux's LVM version 2.02.06, forming a single logical volume. Such configuration provides interesting combinations of QoS attributes with varying attributes of read and write performance as well as reliability. However, while formatting the 4.5 TB filesystem we noticed that running experiments on such a large filesystem could be very time consuming.

We then analysed the measurements for each RAID level. Figure 5.1 illustrates the throughput of sequential reads of varying size across the LVM concatenated filesystem. Figure 5.2 does the same for write operations. Observing the best and worst possible buffer sizes (i.e. those that maximise or minimise the throughput), we calculated the time consumed in reading from or writing to the entire filesystem. Table 5.1 illustrates the results.

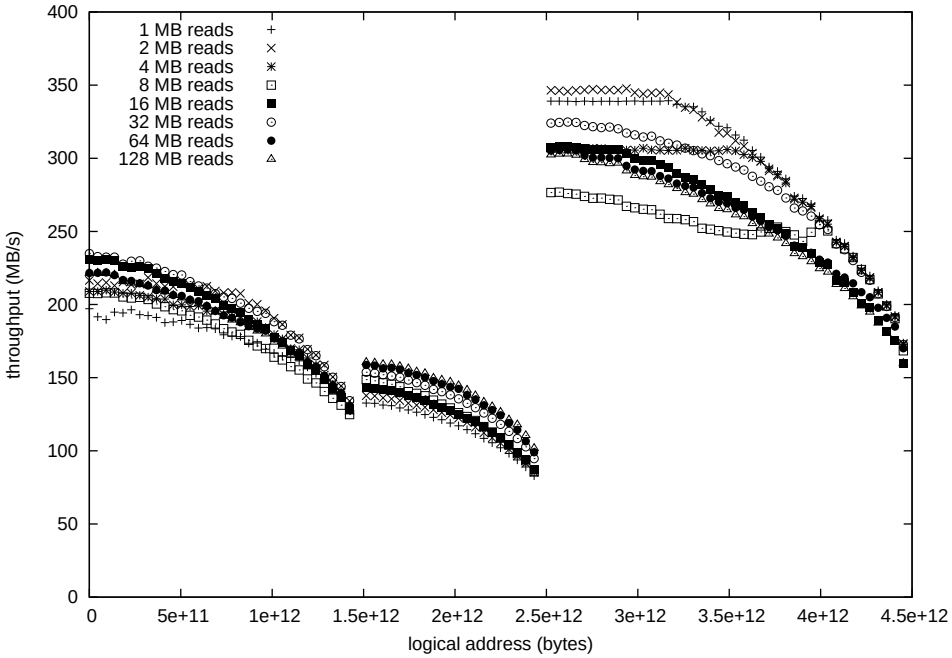


Figure 5.1.: Throughput of sequential reads of varying size across the LVM concatenated filesystem.

Tier	Best Read Time	Worst Read Time	Best Write Time	Worst Write Time
RAID5	7 096 s	8 192 s	8 022 s	36 747 s
RAID10	6 610 s	8 119 s	7 861 s	18 968 s
RAID0	6 518 s	7 908 s	8 668 s	29 259 s
Total	5h 37m 4s	6h 43m 39s	6h 49m 11s	23h 36m 14s

Table 5.1.: Times to read from and write to the proposed tiers' addressable space using varying buffer sizes (best and worst for each tier).

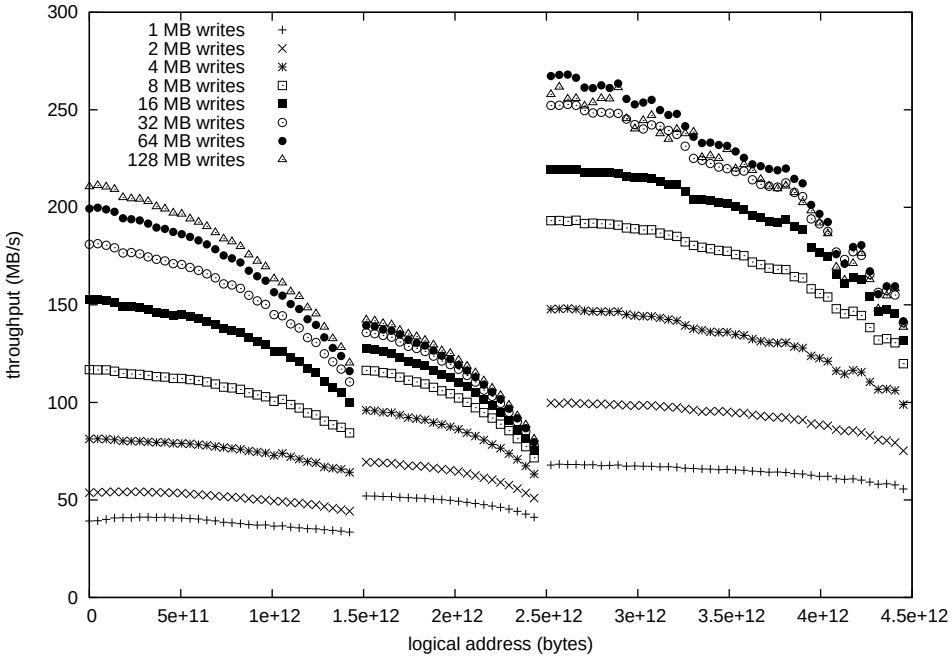


Figure 5.2.: Throughput of sequential writes of varying size across the LVM concatenated filesystem.

After running initial tests, we observed that the time it would take to run some of our experiments was in the order of weeks due to the number of files and directories and the varying buffer sizes. This happens because small write requests cost more than large ones due to latency. In order to work with a more realistic time frame, we therefore created a smaller, more manageable filesystem.

Given that the infrastructure QoS attributes used by our algorithms are obtained through the settings specified in the block groups, we could replicate the original tiers into smaller storage areas without affecting how our algorithms would perceive them. Table 5.2 reflects the proportions of the original infrastructure against our new proposed test environment.

Following this idea, we observed the throughput for each of the listed RAID levels in order to bring forward the appropriate performance QoS attributes. The first step to conduct this analysis was to average the measurements previously illustrated in Figures 5.1 and 5.2. For each operation (read and write), we plotted the throughput average at different positions in the filesystem. Next, we calculated a linear average for each tier.

	Original Infrastructure	New Test Environment
RAID5	1.5 TB	1.5 GB
RAID10	1 TB	1 GB
RAID0	2 TB	2 GB
Total	4.5 TB	4.5 GB

Table 5.2.: Comparison between the capacities of the original storage infrastructure and the test environment.

Figures 5.3 and 5.4 illustrate these values for read and write operations respectively. Based on the linear average, we could classify them as *low*, *medium* or *high* based on the relationship of the performance levels.

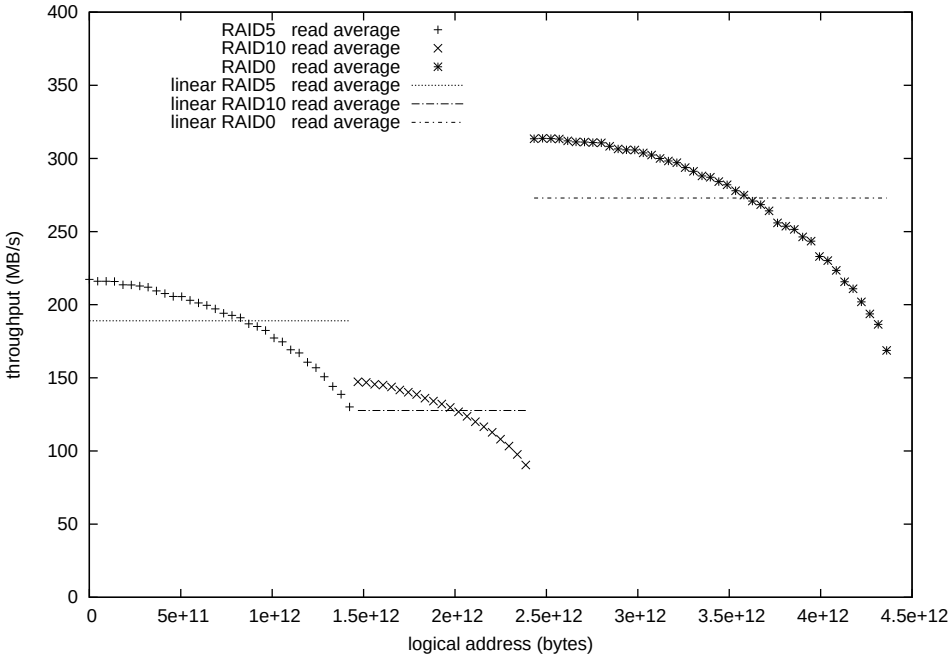


Figure 5.3.: Analysis of average throughput for sequential reads across the LVM concatenated filesystem.

Regarding both the read and write performances we can safely infer that, on average, RAID0 will perform better than RAID5 that, in turn, performs better than RAID01. Similarly, we defined reliability attributes according to the studies presented in Section 2.1; that is, the number of drives that are required to fail in order to occur data loss. While this relative level

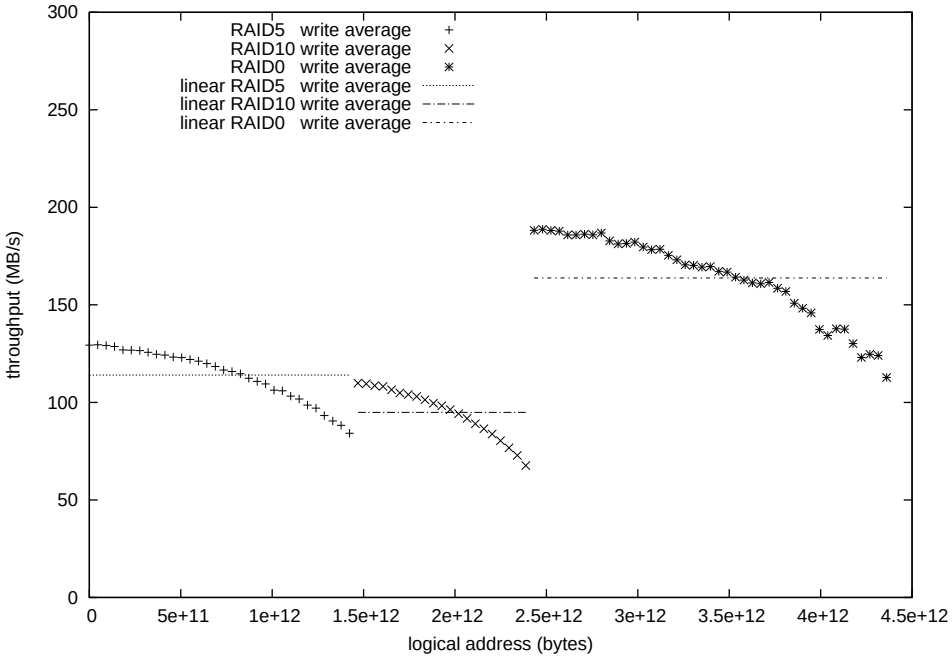


Figure 5.4.: Analysis of average throughput for sequential writes across the LVM concatenated filesystem.

of detail suffices for our experiments, each tier could be broken down into smaller areas (up to the size of a block group) for QoS delivery definitions. Table 5.3 shows the final mapping.

	Read Performance	Write Performance	Reliability
RAID5	Medium	Medium	Medium
RAID10	Low	Low	High
RAID0	High	High	Low

Table 5.3.: Mapping of relative QoS levels to each storage tier.

Unless noted otherwise, the experiments in this chapter were performed on a 4.5 GB filesystem with 4096 bytes datablocks. This configuration provides block groups of 128 MB, thus allowing for a total of 36 block groups. According to the size of each tier in the original infrastructure, we managed to distribute the respective proportion of space efficiency for each RAID group and assign the block group layout accordingly. The resulting distribution is indicated in Table 5.4.

	Capacity	Number of 128 MB Block Groups
RAID5	1.5 GB	12
RAID10	1.0 GB	8
RAID0	2.0 GB	16
Total	4.5 GB	36

Table 5.4.: Test filesystem block group division.

After assigning the corresponding QoS attributes to each block group, we proceed to present our data layout visualisation mechanisms. These tools use the same evaluation techniques as the allocation and migration algorithms, but have added capabilities to allow for the visualisation of the layout improvements provided by the QoS enhancements.

5.2. Data Layout Visualisation Mechanisms

To assess the benefits of using `ext3ipods`, we have developed a suite of tools capable of providing system administrators with insightful information regarding how data is laid out on the filesystem. These tools are based on the same evaluation scheme introduced in Section 4.3 in order to determine how well does a datablock fit in its current location.

5.2.1. Block Group QoS Evaluation

The first evaluation mechanism that was implemented enables the presentation of a numerical score for a given block group. This value allows an instant perception on the QoS-wise allocation quality of the filesystem, but without providing detailed information on each particular QoS attribute. However it is displayed for each block group, the evaluation process must be ran for the entire filesystem due to the addressing scheme of datablocks.

As discussed in Section 3.4 and illustrated further in Figure 4.2, a datablock placed virtually anywhere in the filesystem may be assigned to an inode located in a different block group. Considering the QoS attributes desired by a datablock lies with the inode, it is necessary to scan the entire filesystem in order to properly calculate the allocation score for a block group. Algorithm 4 presents this idea in pseudocode.

Algorithm 4: Evaluation of block group QoS allocation scores.

```
1 BGScore[number_of_block_groups] = array_initialised_with_zeros;
2 foreach inode in filesystem do
3 |   foreach datablock associated with inode do
4 |   |   dbBG = block_group_where_datablock_resides;
5 |   |   bScore = EvalQoS(GroupDsc[dbBG].Delivered,
  |   |   Inode.Required);
6 |   |   BGScore[dbBG] + = bScore;
7 |   endfch
8 endfch
9 return BGScore[];
```

Line 1 of the algorithm initialises with zero the *BGScore*[] array. Each element of this array will represent the QoS evaluated score for the corresponding block group. Next, lines 2 to 8 represent a loop that iterates through all the allocated inodes for the filesystem. It is not necessary to parse other filesystem metadata, as inodes exclusively holds QoS requirement attributes. Finally, for every datablock associated to each inode, lines 3 to 7 represent the loop that calculates the block group where the datablock resides (line 4), evaluates the QoS score for the datablock (line 5) and accumulates the resulting value to the *BGScore*[] array (line 6).

The resulting array, returned in line 9, will contain the allocation score for each block group. This is, in practice, a sum of the scores for every datablock allocated in that block group. In the same way as the evaluation works for an inode, the smaller the score for a particular block group the better the allocation scenario.

This metric was implemented within the `ifm` kernel module and its features are accessible via the `/dev/ifm` character device. It offers insightful information to system administrators, providing a general idea of the QoS allocation quality in different parts of an `ext3ipods` filesystem. It can be used, for example, to quantify if a particular tier of the storage infrastructure is not meeting the expected QoS standards. However, a different mechanism is necessary to evaluate data layouts on an individual QoS attribute basis.

5.2.2. QoS Allocation Bitmaps

The next visualisation mechanism we developed allows for the individual evaluation of each QoS attribute. It consists of a bitmap image for each attribute, where each pixel in the image represents a datablock and its colour is defined by the allocation score. This allows for an immediate perception of the allocation quality on an attribute basis for each block group.

To generate this image, we first use the `ifm` kernel module to retrieve an inode allocation map. This consists of a listing containing the inode number with the corresponding QoS requirements and the allocated datablocks associated to it. Combining this list with the set of QoS attributes delivered by each block group, we are able to infer the evaluation score for a particular allocated position in the filesystem and represent it in the shade of a colour.

Considering the system used in this work allows for three QoS attribute levels (*low*, *medium* or *high*), five possible provisioning situations may arise. These situations are listed in Table 5.5, along with the colour shade we used for each one of them.

Provisioning Scenario	Desired QoS Level	Delivered QoS Level	Colour Shade
Unused Space	N/A	N/A	Grey
Very Underprovisioned	High	Low	Red
Underprovisioned	High	Medium	Yellow
	Medium	Low	
Perfect Match	Low	Low	Green
	Medium	Medium	
	High	High	
Overprovisioned	Low	Medium	Light Blue
	Medium	High	
Very Overprovisioned	Low	High	Dark Blue

Table 5.5.: Possible provisioning scenarios.

Due to different block groups delivering different levels of each QoS attribute in a system, we included a column to the left of the bitmap image indicating the QoS level delivered. Considering we are working with *low*, *medium* and *high* levels, we used the colour shades *red*, *yellow* and *green* respectively to represent the QoS attribute level delivered by a block group. This allows for an easier comprehension of the bitmap.

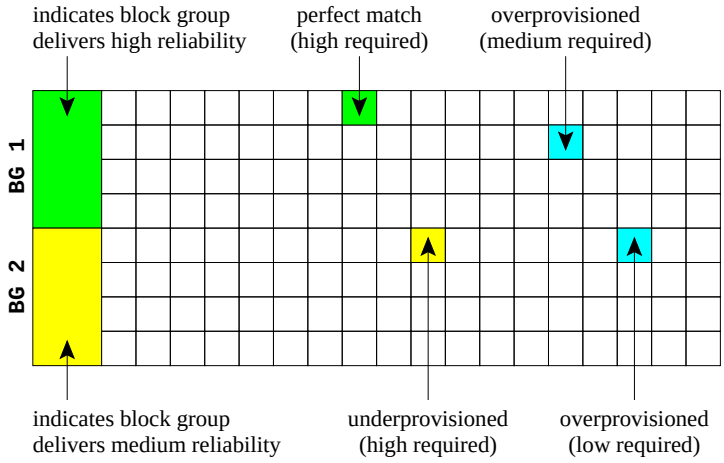


Figure 5.5.: Reliability evaluation example containing two block groups.

Figure 5.5 illustrates an example by showing the allocation bitmap for the reliability QoS attribute in a two block group set. There, BG 1's leftmost column is green, indicating that the block group delivers high reliability. Similarly it is possible to see that BG 2 delivers medium reliability.

Four datablocks appear allocated in this map. Block group BG 1 has one datablock perfectly allocated (green). It is possible to infer that this datablock requires high reliability because this is the QoS level delivered by the block group, thus the only possible combination is a perfect match as indicated in Table 5.5. In the same sense, the light blue datablock must require medium reliability and is therefore overprovisioned.

Regarding block group BG 2, we observe one datablock that also evaluates to light blue. In this case, different from the overprovisioning in BG 1, the datablock must require low reliability. The yellow datablock is underprovisioned and therefore requires high reliability. We stress that this is the same requirement as the green datablock in BG 1, but appears here in yellow due to the different QoS levels delivered by the block groups.

In practice, we developed a bitmap image generation software that receives as input an inode allocation map and the QoS levels delivered by each block group. In the output, each image represents a particular QoS attribute and each pixel in that image (apart from block group division lines and the left column) represents a datablock, with its colour meaning how well it matches that particular attribute's QoS level.

We find that these bitmap allocation images can provide great insight at an individual QoS attribute basis when evaluating a filesystem state. However, to numerically assess the provision factors between desired and delivered QoS, we use an additional mechanism.

5.2.3. Numeric Provisioning Evaluation

The last evaluation mechanism we provide is used to numerically quantify the QoS matching across an entire filesystem. Basically, it computes the percentage of data that ends up in each category of provisioning for each QoS attribute. This is perhaps the most useful tool when comparing the effectiveness of different allocators.

In order to achieve this, we create a set of counters for each QoS attribute. Each counter in the set represents the provisioning scenarios for its attribute; that is, very underprovisioned, underprovisioned, perfectly matched, overprovisioned and very overprovisioned. The idea consists in parsing the same inode allocation list used in the previous evaluation mechanisms. For every datablock read, evaluate the QoS score between the delivered attributes of the block group in which it resides and the desired attributes of the owing inode. Finally, use the computed score to increment the corresponding counters.

After parsing the list, each counter indicates the number of datablocks allocated in that category. Normalising the counters produces the percentage of data in each provisioning category. As will be shown in Section 5.4, this proves an effective way to visualise the benefits of the proposed QoS-aware allocators.

5.3. A QoS-enabled Benchmarking Tool

In order to populate our test filesystem, we required a benchmarking environment capable of generating realistic content. We found that many studies with similar needs [60, 61, 75, 2, 38] used Impressions [3], which is a benchmarking framework capable of generating statistically accurate filesystem images using stochastic distributions. It is then possible to use the same parameters under different environments to recreate particular experiments under the same conditions every time.

Impressions gained attention in the filesystem performance community due to the important goals it aimed at. For once, the work valued the layout of directories and files, including depth of directory trees and variety of file extensions. It showed how these elements played a role on the benchmarking of different applications. As one example, Linux's "find" is heavily affected by a fragmented filesystem or one with deep directory trees. Another example is indexing operations, which is dependent on file types.

To control the variation of these parameters, Impressions offers flexible configuration files that ranges from an automated mode (where a user is only required to provide a size for the filesystem image to populate) to a user-specified mode, where everything can be fine tuned. A list of the parameters that can be configured, alongside their default values, can be found in Table 5.6. Further information on the defaults can be found in [3].

Parameter	Default Model and Parameters
Directory count with depth	Generative model
Directory size (number of subdirs)	Generative model
File size by count	Lognormal-body (α_1=0.99994, μ=9.48, σ=2.46) Pareto-tail (k=0.91, χ_m=512 MB)
File size by containing bytes	Mixture-of-lognormals (α_1=0.76, μ_1=14.83, σ_1=2.35 α_2=0.24, μ_2=20.93, σ_2=1.48
Extension popularity	Percentile values
File count with depth	Poisson (λ=6.49)
Bytes with depth	Mean file size values
Directory size (files)	Inverse-polynomial (degree=2, offset=2.36)
Degree of Fragmentation	Layout score (1.0) or pre-specified workload

Table 5.6.: Impressions configurable parameters with default models.

However ideal as a benchmarking tool for the purposes of this work, Impressions, as released in its website at the time of writing, does not yet support our QoS attribute standards. We therefore studied and enhanced its source code to add QoS extensions for its filesystem populating code.

Following predefined configurations, parameters and stochastic distributions, Impressions will randomly create a set of directories and files in order to populate a filesystem. To include QoS extensions, we have modified its directory creation routine to set a particular set of desired QoS attributes immediately after creating a new directory.

There is a reason why QoS attributes are not also set on the creation of files. This is due to the attribute inheritance feature of `ext3ipods`. By default, `ext3ipods` will set QoS attributes on each newly allocated inode according to the attributes set on its parent directory. Considering this feature, setting desired QoS attributes on new directories alone not only suffices to guarantee that QoS attributes will also be added to files, but actually reinforces the concept that files in the same directory share similar characteristics.

While this feature is essential, considering data is usually directly written to inodes immediately after their allocation[1], there is a case where it may cause a problem. When subdirectories are created, they will inherit their parent's QoS attributes and immediately allocate a datablock to store its entries. If this subdirectory QoS attributes are then modified in order to meet its own files' requirements, its datablocks (i.e. the datablocks containing the directory entries) will probably find themselves misallocated.

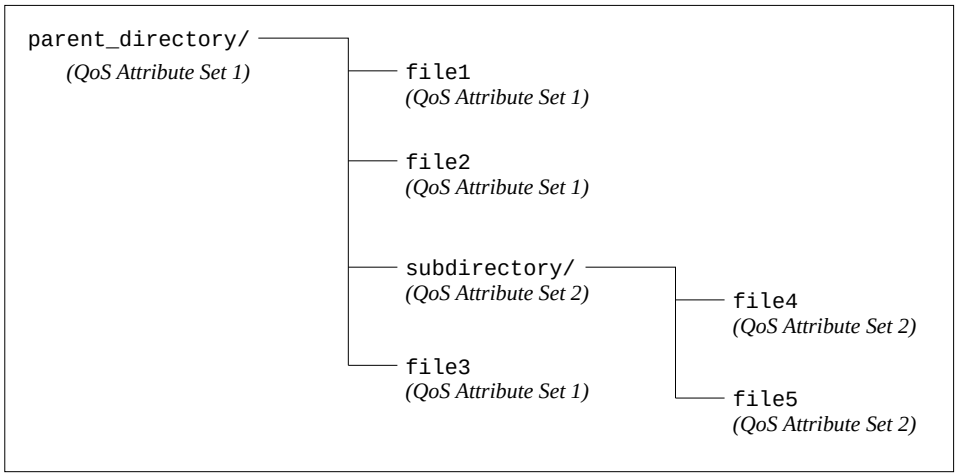


Figure 5.6.: Example of directory tree with associated sets of QoS attributes.

Figure 5.6 illustrates this scenario with a directory tree example. Assuming `parent_directory/` is created with a QoS attribute Set 1, every inode allocation made inside it will inherit the same QoS attribute set. This includes files 1 to 3 and `subdirectory/`, all which will be allocated in order

[1]Most applications will create an inode and immediately start writing data to it (e.g. Unix's `/bin/cp`). One way to set QoS attributes prior to writing data, thus allowing a QoS-enabled datablock allocator to perform properly, would be to modify existing applications to set QoS flags after creating a new inode, but before writing to it.

to meet Set 1. If files 4 and 5 are supposed to be allocated with QoS attribute Set 2, the attribute set of `subdirectory/` must first be changed accordingly. This, however, is likely to cause the subdirectory's datablocks to be misallocated until a migration is performed.

In order to prevent Impressions from migrating every new subdirectory created that does not have the same QoS requirements of its parent, we have adjusted the QoS attribute setting strategy. Firstly, we modify the parent's attribute set to match the set of the subdirectory to be created. Next, we create the subdirectory (causing it to be allocated with the correct attribute set) and only then restore the parent's original attribute set.

It is important to understand that this approach may present race conditions in scenarios with concurrent allocations. However, the present work is limited to studying the benefits of QoS-aware data placement and thus a locking scheme to resolve these race conditions lies outside the scope of our present work.

Regarding which QoS attribute levels to use, two possibilities were considered. The first involves using a set containing all possible combinations of *low*, *medium* and *high* levels of reliability and read and write performance. The second consists of using only the combination of attribute levels that are supported in the infrastructure, as defined in Table 5.3. Both cases will be considered in Section 5.4.

Finally, we evaluated how to select which QoS attribute combination to set on a newly created directory. Again, two possibilities were considered. The first involved selecting attributes using a robust random number generator such as the Mersenne Twister [68]. The second possibility consisted in simply choosing QoS sets on a rotating basis, looping back to the first combination used once all of them have been applied. Due to Impressions already using parameterised distributions to determine when to create directory or files, we found that the latter alternative was sufficient for our purposes.

5.4. Scenarios and Results

Having defined a test environment, data layout evaluation strategies and a benchmarking framework to populate filesystems, this section presents experiments and results with different scenarios. We start by populating

our test filesystem with the default `ext3fs` allocators to collect metrics and establish a baseline for comparison. Next, we repeat the experiments using variations of our QoS-enhanced algorithms. Finally, we compare the results.

5.4.1. Baseline Establishment

To establish a baseline for comparison, we defined Impressions to populate about 4.0 GB (around 90%) of the test filesystem using its default settings and the `ext3fs` allocators. These numbers provide enough data to stress the filesystem occupancy in a meaningful manner. Given the stochastic nature of the distributions used (which are based on random numbers) and issues such as internal fragmentation, we observed that about 96% of the filesystem was populated in practice. Considering we will use the same parameters for the next experiments, the same write operations, in the exact same order, will be executed. While the disk occupancy will be the same, the data layout should be different due to the allocators used.

The initial evaluation, which shows the score for each block group, produced the values listed in Table 5.7. However, these values alone cannot be used to infer how good or bad the allocation is. To assess if the layout has improved or deteriorated QoS-wise in a real world situation, they must be compared to the same filesystem, with the same allocators, of a different moment in time. Alternatively, as is the case with our test environment, they can be compared to the exact same benchmarking procedure, but using a different allocation algorithm. As an observation on the results, it is likely that block group 18 scored 0 because no data has been allocated to it. A more thorough investigation of the allocation map confirmed this assessment to be correct.

Figures 5.7, 5.8 and 5.9 display the allocation bitmaps for the reliability, read and write performance respectively. It is possible to confirm that block group 18 was indeed unpopulated (visible through the grey area in the maps) due to the characteristics of the stock allocator, justifying the score of 0 observed earlier in Table 5.7. These bitmaps reflect the QoS unawareness of the `ext3fs` default allocator by showing a large number of misallocated datablocks. However, in the same way as the first results obtained, these bitmaps can be better analysed when compared to the same scenario of a different time or to the same procedure when using different allocators.

Tier	Block Group Index	Block Group Score	Accumulated Tier Score
RAID5	1	99 438	891 969
	2	87 070	
	3	89 870	
	4	106 068	
	5	75 599	
	6	88 833	
	7	73 700	
	8	127 846	
	9	84 354	
	10	5 327	
	11	18 701	
	12	35 163	
RAID10	13	135 010	876 465
	14	172 917	
	15	171 160	
	16	137 153	
	17	151 980	
	18	0	
	19	27 660	
	20	80 585	
RAID0	21	5 260	1 740 244
	22	119 296	
	23	81 008	
	24	131 056	
	25	93 288	
	26	129 936	
	27	83 876	
	28	127 840	
	29	128 980	
	30	121 096	
	31	119 236	
	32	128 968	
	33	129 040	
	34	129 008	
	35	115 100	
	36	97 256	

Table 5.7.: Block group evaluation scores whilst using default allocators.

Figure 5.7.: Default `ext3fs` allocator bitmap for the reliability QoS attribute.

Figure 5.8.: Default `ext3fs` allocator bitmap for the read performance QoS attribute.

Figure 5.9.: Default `ext3fs` allocator bitmap for the write performance QoS attribute.

Finally, we quantify the allocation scenario numerically using our third evaluation mechanism. Table 5.8 lists the number of datablocks encountered under each provisioning category. Additionally, the last two rows also shows the total datablock count and the respective percentage of occupation.

Despite the three evaluation summaries displayed in this section being better interpreted when compared at different moments in time, allowing for an understanding of how the data layout has evolved QoS-wise, they already provide some insight on the performance of the default `ext3fs` allocators. Especially on the results presented in the allocation bitmaps and in Table 5.8, it becomes evident that using QoS-unaware allocators cause data layouts to appear in a chaotic manner when it comes to QoS evaluation.

This further reinforces the fact that a different approach must be obtained when it comes to the existing solutions listed in Chapter 2. This means that, even after rearranging infrastructure layouts or migrating data to different locations in order to meet QoS requirements, QoS-unaware allocators will continue to focus on the wrong aspects and disturb the storage layout.

	Very Underprov.	Underprov.	Perfectly Matched	Overprov.	Very Overprovisioned
Reliability	62 397	420 378	390 711	152 282	56 677
Read Performance	56 672	152 207	390 716	420 453	62 397
Write Performance	56 672	152 207	390 716	420 453	62 397
Total Count	175 741	724 792	1 172 143	993 188	181 471
Average	5.41%	22.32%	36.10%	30.58%	5.59%

Table 5.8.: Datablock count under each provisioning category for the default `ext3fs` allocator.

However it is out of the scope of this research to evaluate the performance of the implemented algorithms, we have annotated the time it took to run each experiment. These values were summarised at the end of this chapter on Section 5.5.

5.4.2. Evaluation of the `ext3ipods` inode QoS-aware Allocator

Following the experiments conducted with the stock `ext3fs` allocator, we continue to test the `ext3ipods` QoS-aware inode allocator. This is done by initialising another copy of the 4.5 GB filesystem with the same block group settings, but reloading the kernel module with the new inode allocator. Next, we populate the filesystem using the same settings in Impressions. This will take the filesystem to the same occupancy level as in the previous test; that is, 96%.

At this point, we expect to see a meaningful improvement on the filesystem layout. This is mainly because the stock `ext3fs` datablock allocator will attempt to find datablocks within the same block group as the inode itself when there is no data allocated for that inode. If there was data previously allocated to the inode, then the `ext3fs` allocator would consider the block group of the last allocated datablock.

Table 5.9 lists the results showing block group evaluation scores for the first evaluation mechanism. Being aware that the filesystem is occupied to 96% of its capacity, these results already show a dramatic improvement. This is inferred by the QoS scores of the block groups in the RAID5 tier, which indicates a flawless allocation QoS-wise. The following tier shows a score of 560, which is a great improvement on the 876 465 assessed with the stock `ext3fs` allocator. The last tier scored 895 924, which may appear high but is under half of the 1 740 244 obtained scored in the first experiment.

Analysing the allocation maps for this second experiment, we present Figures 5.10, 5.11 and 5.12 illustrating the QoS match of the reliability, read and write performances respectively. Again, it is possible to infer that there is a large amount of perfectly matched data due to green areas visible in the map. Apart from the reliability map, we also note blue sections on the RAID0 tier. This contributes to the cause of the higher scores observed for that tier on the first evaluation mechanism.

Due to the homogeneous allocation especially on the first tier, careful analysis will allow the observation of a continuous set of grey datablocks in the beginning of each block group. We stress that this is not free space. Rather it is the block group metadata as described in the filesystem layout and illustrated in Figure 3.7.

Tier	Block Group Index	Block Group Score	Accumulated Tier Score
RAID5	1	0	0
	2	0	
	3	0	
	4	0	
	5	0	
	6	0	
	7	0	
	8	0	
	9	0	
	10	0	
	11	0	
	12	0	
RAID10	13	60	560
	14	36	
	15	48	
	16	6	
	17	365	
	18	0	
	19	25	
	20	20	
RAID0	21	0	895 924
	22	0	
	23	0	
	24	0	
	25	0	
	26	68 648	
	27	22 196	
	28	99 784	
	29	129 004	
	30	129 008	
	31	86 652	
	32	57 472	
	33	28 896	
	34	152 400	
	35	121 864	
	36	0	

Table 5.9.: Block group evaluation scores whilst using the QoS-aware inode allocator.

Continuing to analyse the allocation maps, we observe that the small second tier score that was listed in Table 5.9 refers to some reliability overprovisioning and a minor underprovisioned area in the read and write performance maps. This occurs due to the high occupancy of the filesystem and the exhaustion of space for perfect matches.

Finally, on the last block groups of the last tier, we see a large underprovisioned area in the reliability map and a large overprovisioned area in the read and performance maps. Combining the facts that the occupancy of the filesystem is high and that this misallocation happened towards the end of the tier, we theorise that this event happened due to the lack of free space in order to meet the QoS requirements of the data written last. Indeed, the last tier offers the largest low reliability area and therefore the hardest criteria to meet, considering Impressions will, to some extent, evenly choose between the three combinations of QoS levels as discussed in Section 5.3.

To support this conjecture, we reexecuted this second experiment, but configuring Impressions to allocate only 2.0 GB of the 4.5 GB filesystem. We then expected to see a smaller allocated area in each tier, as well as no underprovisioning cases whatsoever. Appendix A.1 confirms this conjecture by presenting and discussing additional results. Additionally, we realised that apart from the three possible QoS attribute combinations, files created in the root directory of the filesystem were set to the same attributes as the root directory itself. Considering we decided upon not setting any requirements for the root directory, these files are allocated based on a low reliability and low read and write performance QoS requirements. This will lead to overprovisioning at some extent, because our infrastructure does not have any block groups delivering such combination of QoS attributes.

Finally, we obtain the datablock QoS provisioning match count. Table 5.10 lists these results, reflecting the allocation maps previously analysed. We observe that the numbers concentrated in the Perfectly Matched section, improving from 36.10% when using the `ext3fs` allocators to 81.71% under the new scenario. Other notable numbers are the reduction in underprovisioning, that accumulated 27.73% in the previous test and improved to 6.1% and the overprovisioning, that improved from 36.17% to 12.61%.

Figure 5.10.: QoS-aware inode allocator bitmap for the reliability QoS attribute.

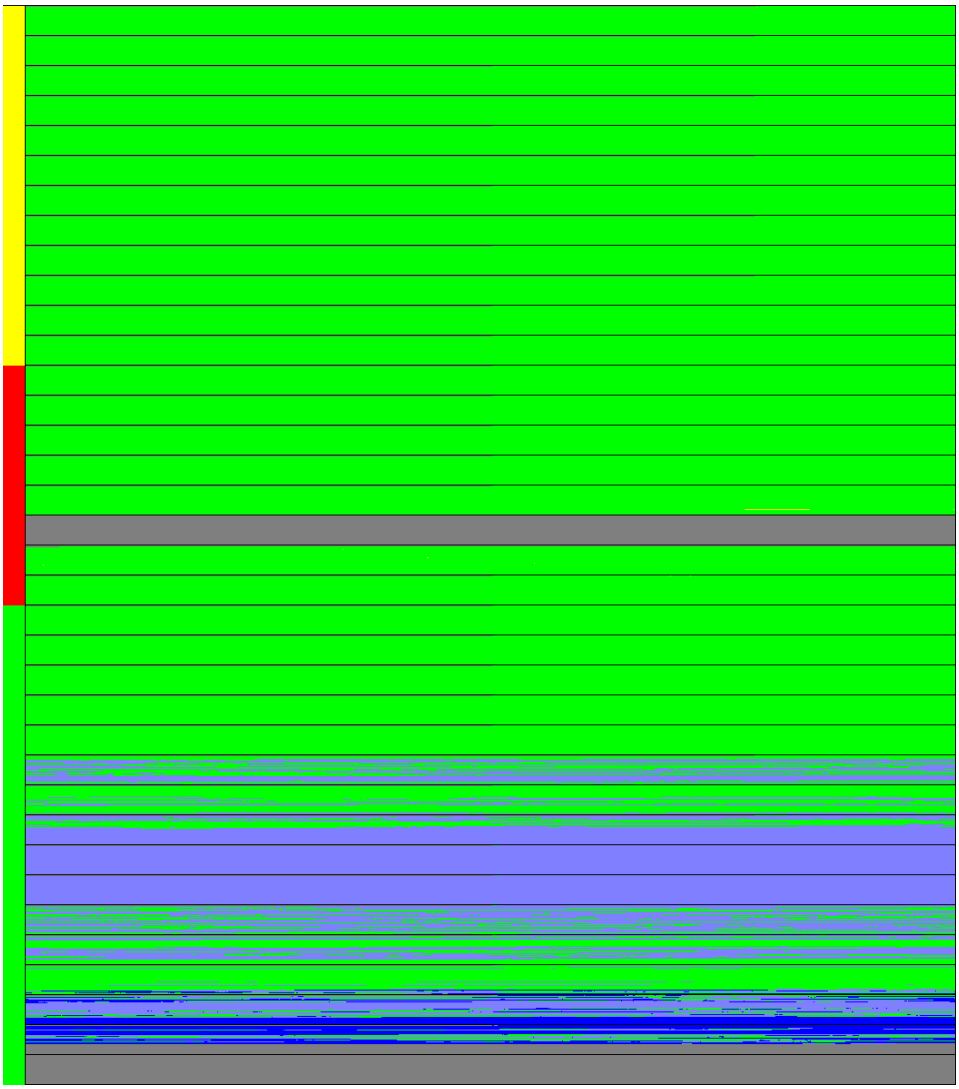


Figure 5.11.: QoS-aware inode allocator bitmap for the read performance QoS attribute.

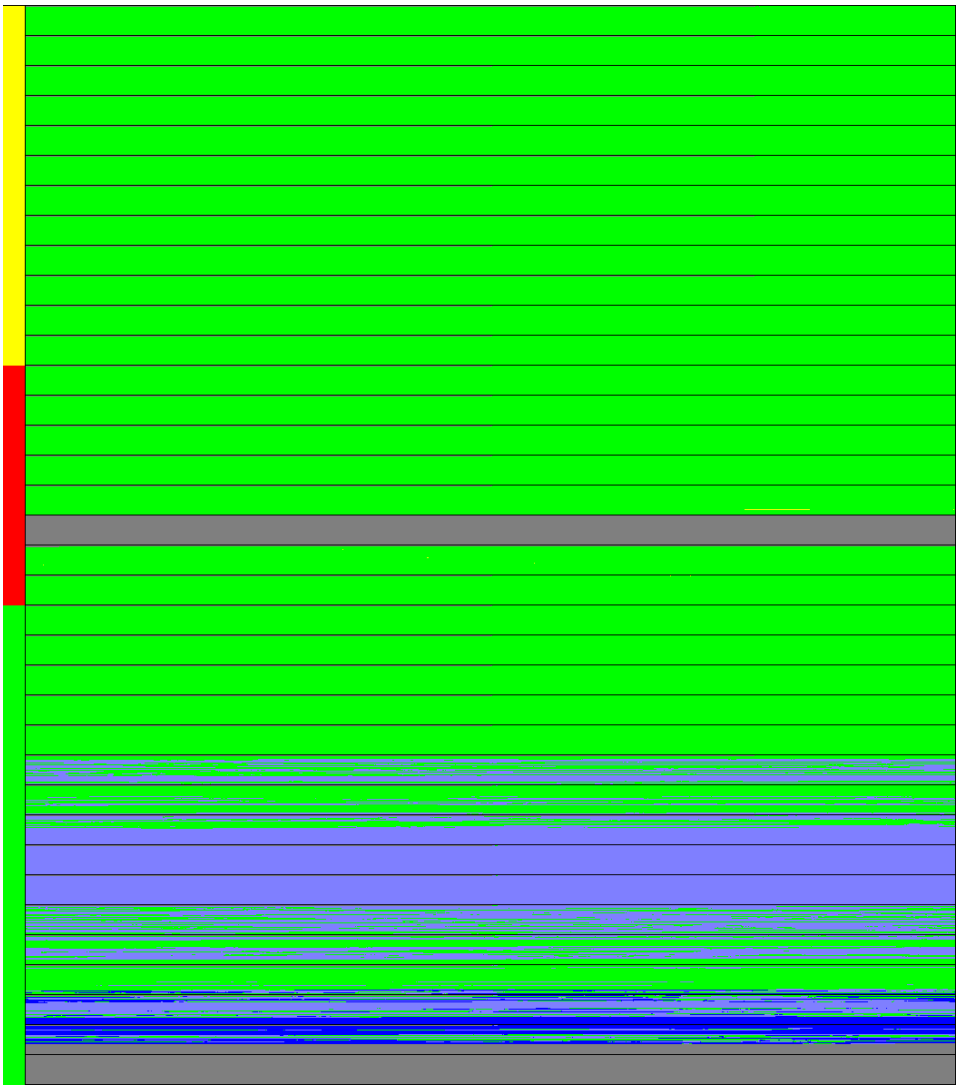


Figure 5.12.: QoS-aware inode allocator bitmap for the write performance QoS attribute.

	Very Underprov.	Underprov.	Perfectly Matched	Overprov.	Very Overprovisioned
Reliability	26 147	171 687	884 451	80	80
Read Performance	0	80	884 531	171 687	26 147
Write Performance	0	80	884 531	171 687	26 147
Total Count	26 147	171 847	2 653 513	343 454	52 374
Average	0.81%	5.29%	81.71%	10.58%	1.61%

Table 5.10.: Datablock count under each provisioning category for the QoS-aware inode allocator.

5.4.3. Evaluation of the `ext3ipods` Datablock QoS-aware Allocator

The last and most sophisticated allocator we evaluate is the `ext3ipods` datablock QoS-aware allocator. This adds to the `ext3ipods` QoS-aware inode allocator by evaluating the datablock allocation requests with regards to the match of QoS attributes. It is important to stress that this allocator encompass the previously experimented allocator, also considering the QoS elements for the allocation of inodes. Similarly to the previous example, we initialise a separate copy of the 4.5 GB test filesystem, set the block group QoS delivery attributes and run the QoS enhanced version of Impressions with the same set of parameters.

Last experiment produced results showing that our approach is very efficient when it comes to allocate inodes considering QoS elements. For this experiment, however, we expect only but a minor improvement on the numbers already seen, with perhaps no noticeable changes to the allocation maps. This is due to the datablocks being initially allocated to the same block group as their inode (which was already allocated following QoS elements) and subsequently allocated following the last datablock.

Following this logic, no improvements are all were to be expected. However, due to the low space conditions imposed by this test, we still expect improvements to some extent. The major gains proposed by this allocator would be visible on cases where previous datablocks (or inodes) were allocated without the proper QoS considerations. Also, when there are QoS attribute changes during the lifetime of data in the system.

Table 5.11 lists the initial assessment of the results obtained in the experiment. We can see an improvement, although relatively small, to the results obtained with the `ext3ipods` QoS-aware inode allocator. The second tier has score improved from 460 to 160, approaching 0. The third tier improved from 895 924 to 895 604. We can safely assume that the score remains high due to same reasons described in the previous section and tested in Appendix A.1.

Tier	Block Group Index	Block Group Score	Accumulated Tier Score
RAID5	1	0	0
	2	0	
	3	0	
	4	0	
	5	0	
	6	0	
	7	0	
	8	0	
	9	0	
	10	0	
	11	0	
	12	0	
RAID10	13	10	160
	14	0	
	15	0	
	16	6	
	17	48	
	18	0	
	19	36	
	20	60	
RAID0	21	0	895 604
	22	121 864	
	23	152 400	
	24	28 256	
	25	57 792	
	26	85 476	
	27	129 016	
	28	127 860	
	29	101 096	
	30	22 900	
	31	68 944	
	32	0	
	33	0	
	34	0	
	35	0	
	36	0	

Table 5.11.: Block group evaluation scores whilst using the QoS-aware datablock allocator.

Next, we generate the allocation maps and present Figures 5.13, 5.14 and 5.15 indicating reliability, read and write performance respectively. Although they look similar to the results of the previous section, we observe minor differences regarding the location of under and overprovisioned areas.

Perhaps the most noticeable discrepancy is the location of the very underprovisioned datablocks that appear on the third tier. In the previous experiment, they appear towards the end of the tier. In Figure 5.13, however, they appear on the first block groups of the tier. This event is explained due to the implementation of the datablock QoS-aware allocation algorithm.

When the algorithm was implemented, we used insertion sort to keep the priority queue sorted according to the QoS score of the block groups. This means that while the block groups were scanned linearly from first to last, the best matches were always inserted in the beginning of the queue, causing the last located block groups to be the first candidates for allocation.

To further illustrate the block groups being selected from last to first from the priority queue, we used the set of Impressions's parameters that cause it to populate only 2 GB of the test filesystem. The allocation bitmaps for this additional experiment are listed in Appendix A.2 and can be compared to those of Appendix A.1, where the default `ext3fs` allocators were used.

For the last analysis, we obtain the datablock QoS provisioning match count. Table 5.12 lists these results, again reflecting the allocation maps previously analysed. We observe that the numbers concentrated even further in the Perfectly Matched section, improving from the original 36.10% when using the `ext3fs` allocators to 81.72% under the new scenario. The overall underprovisioning reduced from 27.73% in the original test to 6.09%, while the overall overprovisioning improved from 36.17% to 12.19%.

5.5. Final Considerations

This chapter presented a series of experiments conducted on copies of a test filesystem in order to show the benefits of QoS-aware allocation algorithms. Comparing the results obtained in Section 5.4.1, 5.4.2 and 5.4.3, we can observe the efficiency of our proposed relative QoS score evaluation formula. To further emphasise the improved placement provided by our QoS-aware solutions, Figure 5.16 presents a histogram comparing the provisioning factors achieved by each allocator.

Figure 5.13.: QoS-aware datablock allocator bitmap for the reliability QoS attribute.

Figure 5.14.: QoS-aware datablock allocator bitmap for the read performance QoS attribute.

Figure 5.15.: QoS-aware datablock allocator bitmap for the write performance QoS attribute.

	Very Underprov.	Underprov.	Perfectly Matched	Overprov.	Very Overprovisioned
Reliability	26 067	171 767	884 531	0	80
Read Performance	0	0	884 611	171 767	26 067
Write Performance	0	0	884 611	171 767	26 067
Total Count	26 067	171 767	2 653 753	343 534	52 214
Average	0.80%	5.29%	81.72%	10.58%	1.61%

Table 5.12.: Datablock count under each provisioning category for the QoS-aware datablock allocator.

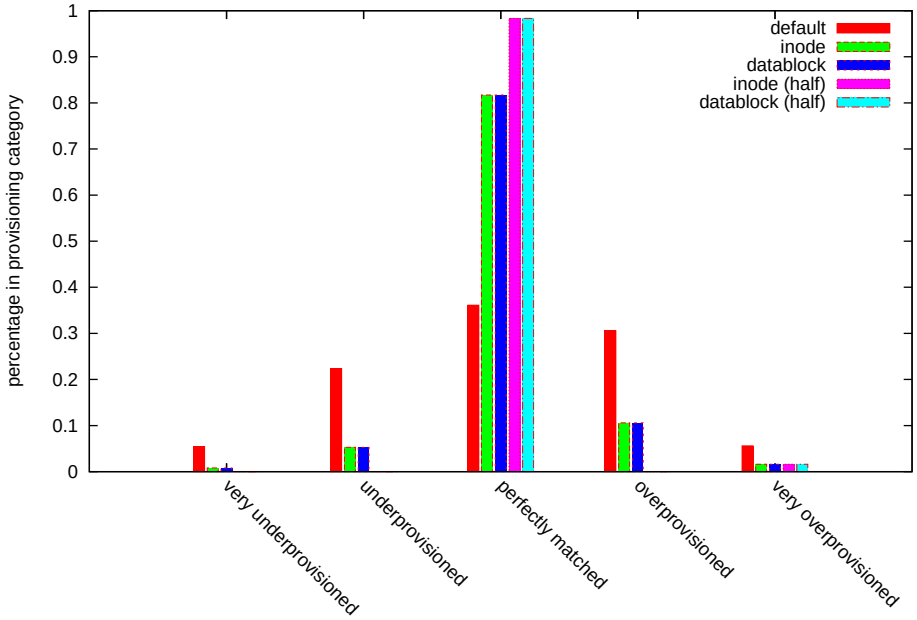


Figure 5.16.: Comparison of the provisioning obtained with different allocators.

The histogram plots five different experiments. Firstly, we see in red the provisioning obtained by the default `ext3fs` allocator. Next, we see the `ext3ipods` inode and datablock QoS-aware allocators in green and blue, respectively. Finally, we see in purple and cyan the additional results of the same QoS enhanced allocators when executed on a half populated filesystems, showing their increased efficiency given enough storage space.

Regarding the performance of the implemented algorithms, we have annotated the time to execute the experiments using each set of allocators. These are summarised in Table 5.13, including the total time to populate the filesystem and their relative performance to the default `ext3fs` allocators.

While Figure 5.16 shows the layout of the filesystem is improved significantly with the proposed strategies, Table 5.13 shows that the implementation does not perform as well as the original `ext3fs` allocators. We note that this measurement evaluates only the time to find new inodes and datablocks for allocation. The actual usage performance of the filesystem will reflect the QoS match in terms of desired and delivered performance attributes. Also, alternatives for improving these implementations in terms of performance were discussed in Section 4.5.

Allocator	**Time to Populate**	**Relative Performance**
Default `ext3fs` Allocator	108 secs	1.00
QoS-aware inode Allocator	271 secs	0.40
QoS-aware Datablock Allocator	293 secs	0.37

Table 5.13.: Time to populate the 4.5 GB test filesystem.

6. Conclusions and Future Work

While the growth in the amount of data managed by organisations shows no signs of abating, the human resources involved in the managing of storage volumes also increases. This has driven research towards automated solutions that attempt to analyse I/O requests (considering aspects such as I/O frequency and distribution of read/write buffer sizes) and adjust the infrastructure layer to improve performance.

However, existing approaches do not cater for QoS attributes that cannot be inferred by mere workload analysis, such as reliability. They are also ineffective for cases such as database transactions that need to be executed as fast as possible once invoked, but that may use tables that are not accessed very often. On such automatic systems, these tables would likely occupy non-performatic storage areas.

6.1. Summary of Achievements

This thesis has presented a different approach where, on one hand, QoS requirements of datasets are specified by users and applications and, on the other, QoS attributes delivered by the storage infrastructure are profiled and adjusted by system administrators. With this information, an intelligent filesystem fabric is capable of placing data in order to obtain a good match between desired and delivered relative QoS.

We have prototyped this idea in a working environment by enhancing the popular Linux Extended 3 Filesystem with QoS extensions. Because our implementation makes use of reserved space on the original filesystem structure, an existing `ext3fs` image may be mounted back and forth (with our modules or the stock kernel) without any conversion. Furthermore, we have designed and implemented working QoS-aware allocation algorithms.

In order to develop such algorithms, we first studied the means to compute the match between desired and delivered QoS attributes. This was realised

in the form of a formula that takes these values, for any number of attributes, and provides a score regarding the degree of QoS match between both. The resulting score is used by allocation and migration algorithms as well as evaluation techniques that are used to assess data layouts.

The specification of desired QoS attributes on the datasets has been implemented in the reserved space within inode flags. This allowed us to use existing mechanisms for viewing and changing such flags using the `lsattr` and `chattr` tools respectively, in the same way `chmod` is used to manage an inode's permissions. For cases where this metadata exceeds the available reserved space we proposed the usage of extended attributes, which is another convenient existing mechanism.

Regarding the delivered QoS attributes of the storage tiers, we developed a kernel extension in the form of a loadable module called the iPODS Filesystem Manager (`ifm`). This module, amongst other functions, allows system administrators and profiling tools to conveniently view and update the metadata in a filesystem's block groups through read and write operations on a character device.

Another achievement is the migration algorithms proposed to cater for the evolution of QoS requirements and the continual improvement of the degree of QoS matching. The concepts behind these algorithms are inspired by the online defragmentation mechanism of `ext4fs` and implemented in a similar way. This includes an extension to the `ioctl()` kernel interface, allowing for an elegant implementation of both active and passive strategies.

We enhanced Impressions, a framework for filesystem benchmarking, in order to define different combinations of desired QoS attributes on datasets during the population of a test filesystem. That allowed us to experiment with the default `ext3fs` allocators and our QoS-aware allocators for comparison. We could then use `ifm` to collect data from populated filesystems and show both quantitatively and visually the benefits of the proposed solution.

Finally, we intend to make the source code for the implementations of this work available online. This includes Linux kernel patches for `ext3ipods`, the filesystem management loadable module `ifm`, the suite of visualisation tools and the QoS-enhanced version of `e2fsprogs`. Further information can be found at `http://www.paradoxo.org/`.

6.2. Applications

As discussed in the introduction of this thesis, the usage of multi-tier Virtualised Storage Systems continues to grow in the industry. Due to rapidly-changing business needs such as the implementation of new technologies or features for power efficiency and green computing, to name but a few, the management of Quality of Service in storage infrastructures remains a challenge for every corporation.

While current solutions for the automation of these management tasks focus mainly on improving VSSs performance, our proposal creates new possibilities in the field. By allowing system administrators to specify what their storage infrastructure is capable of delivering and matching such capabilities to the QoS desired by datasets as defined by users and applications, we approach the problem from a complete different perspective.

Following up with this idea, the concepts introduced in this thesis offer a new solution for problems affecting any organisation using a multi-tiered storage infrastructure. This includes banks, media agencies, internet service providers and data centres to name but a few. Although we have implemented our prototype over `ext3fs`, such organisations can apply the same principles to any other filesystem or storage technology.

Furthermore, the presented principle of relative QoS matching can be applied to other problems outside of the storage domain. Many areas requiring the match of two technologies based on relative attributes could benefit from this idea. This includes the selection of iterative solutions for mathematical problems, the migration and placement of virtual machines in hardware providing specific characteristics and the division of personnel for the execution of particular tasks, to name but a few.

6.3. Future Work

There are a number of possible extensions to the work presented in this thesis. Considering we focused on showing the benefits of a QoS-aware allocation strategy for the placement of data in multi-tier VSSs, some ancillary aspects that could have been analysed were put aside for the moment. Apart from subjects that would involve prototyping and experimenting for result analysis, we will also list topics that could be open for future debate.

The first extension we propose is the assessment of external fragmentation when using the algorithms introduced in this thesis. Fragmentation plays an important role in the access performance of data. This is especially true when it comes to rotating magnetic storage, where seek times can represent a considerable part of the access times for data that cannot be contiguously read from the media.

Considering the allocation algorithms were heavily modified to attain the objectives in view, the original principles of preventing filesystem fragmentation were disregarded. To compare the presence of fragmented data, we suggest similar comparisons of our algorithms to the `ext3fs` counterparts under different benchmarking routines. Tests that stress fragmentation include interleaved truncation and later allocation of data in existing files or the concurrent writing of large filesystem entries.

Another extension which would be valuable is the performance evaluation and improvement of the algorithms proposed. While we already indicated possible enhancements in Sections 4.5.2, 4.5.3 and 4.6.3, there was no opportunity to implement and experiment with them. While we emphasise that the focus of this work is to show the benefits of the solution presented, the adoption of such solution is dependent on its performance impact.

This could be achieved in separate stages. Firstly, a study on the performance comparison of the current implementation in `ext3ipods`. These results would provide a crucial baseline for future evaluation of any alterations to the algorithms. Secondly, the implementation of the proposed caches and indices as proposed in the aforementioned sections. Finally, an evaluation and reassessment of the results, possibly enlightening further changes and modifications that are not as tangible at this stage.

With regards to the applicability of this work to a cloud-like environment, we believe there is ground for the exploration of a cost or pricing mechanism. This would apply especially for high levels of QoS attributes, protecting a system from abuse when users or applications require high performance, for example, to all of their data.

In such a cloud environment, this scenario could arise due to the requester for a particular QoS level being unaware regarding the needs of other users in the system. This leaves space for further research on how a quota-like system could be applied. Such mechanism would ensure that users or applications would be required to balance the relative QoS requirements for their data.

Bibliography

[1] The Btrfs Wiki, March 2010. Available at: `http://btrfs.ipv5.de`. Last accessed on February 2012.

[2] Ian F. Adams, Mark W. Storer, and Ethan L. Miller. Analysis of Workload Behavior in Scientific and Historical Long-Term Data Repositories, March 2011. Technical Report UCSC-SSRC-11-01.

[3] Nitin Agrawal, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Generating Realistic Impressions for File-System Benchmarking. In *FAST'09: Proccedings of the 7th Conference on File and Storage Technologies*, pages 125–138, San Francisco, CA, USA, February 2009. USENIX Association.

[4] Sedat Akyürek and Kenneth Salem. Adaptive Block Rearrangement. *ACM Transactions on Computer Systems*, 13(2):89–121, May 1995.

[5] Eric Anderson. Simple Table-based Modeling of Storage Devices, July 2001. HP Laboratories. Technical Report HPL-SSP-2001-4. Available at: `http://www.hpl.hp.com/research/ssp/papers/HPL-SSP-2001-04.ps`. Last accessed on October 2011.

[6] Eric Anderson, Joseph Hall, Jason D. Hartline, Michael Hobbs, Anna R. Karlin, Jared Saia, Ram Swaminathan, and John Wilkes. An Experimental Study of Data Migration Algorithms. In *WAE'01: Proceedings of the 5th International Workshop on Algorithm Engineering*, pages 145–158, London, UK, August 2001. Springer-Verlag.

[7] Eric Anderson, Michael Hobbs, Kimberly Keeton, Susan Spence, Mustafa Uysal, and Alistair Veitch. Hippodrome: Running Circles Around Storage Administration. In *FAST'02: Proceedings of the 1st USENIX Conference on File and Storage Technologies*, pages 175–188, Monterey, CA, USA, January 2002. USENIX Association.

[8] Eric Anderson, Michael Hobbs, Kimberly Keeton, Susan Spence, Mustafa Uysal, and Alistair Veitch. Hippodrome: Running Circles Around Storage Administration. In *FAST'02: Proceedings of the 1st USENIX Conference on File and Storage Technologies*, pages 175–188, Monterey, CA, USA, January 2002. USENIX Association.

[9] Eric Anderson, Susan Spence, Ram Swaminathan, Mahesh Kallahalla, and Qian Wang. Quickly Finding Near-optimal Storage Designs. *ACM Transactions on Computer Systems*, 23(4):337–374, November 2005.

[10] Eric Anderson, Susan Spence, Ram Swaminathan, Mahesh Kallahalla, and Qian Wang. Quickly Finding Near-Optimal Storage Designs. *ACM Transactions on Computer Systems*, 23(4):337–374, November 2005.

[11] Storage Networking Industry Association. Definition of raid level 6, September 2011. Available at: `http://www.snia.org/education/dictionary/r#raid_6`. Last accessed on October 2011.

[12] R. Bayer and E. McCreight. Organization and Maintenance of Large Ordered Indices. In *SIGFIDET'70: Proceedings of the 1970 ACM SIGFIDET Workshop on Data Description, Access and Control*, pages 107–141, New York, NY, USA, November 1970. ACM.

[13] Medha Bhadkamkar, Jorge Guerra, Luis Useche, Sam Burnett, Jason Liptak, Raju Rangaswami, and Vagelis Hristidis. BORG: Block-reORGanization for Self-optimizing Storage Systems. In *FAST'09: Proccedings of the 7th USENIX Conference on File and Storage Technologies*, pages 183–196, San Francisco, CA, USA, February 2009. USENIX Association.

[14] M. Blaum, J. Brady, J. Bruck, and J. Menon. EVENODD: An Optimal Scheme for Tolerating Double Disk Failures in RAID Architectures. In *ISCA'94: Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 245–254, Chicago, IL, USA, April 1994. IEEE.

[15] Henry Bond, Nicholas J. Dingle, Felipe Franciosi, Peter G. Harrison, and William J. Knottenbelt. Data Placement and Migration Strategies

for Virtualised Data Storage Systems. In *ESM'09: Proceedings of the 23rd Annual European Simulation and Modelling Conference*, pages 231–237, Leicester, UK, October 2009. Eurosis.

[16] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly, Sebastopol, CA, USA, 3rd edition, November 2005.

[17] Mingming Cao, Theodore Y. Ts'o, Badari Pulavarty, Suparna Bhattacharya, Andreas Dilger, and Alex Tomas. State of the Art: Where We Are With the ext3 Filesystem. In *Proceedings of the 2005 Linux Symposium*, pages 69–96, Ottawa, Canada, July 2005.

[18] Brian Carrier. *File System Forensic Analysis*. Addison-Wesley Professional, 1st edition, March 2005.

[19] Brian Carrier. The Sleuth Kit, June 2011. Available at: `http://www.sleuthkit.org`. Last accessed on October 2011.

[20] Feng Chen, David A. Koufaty, and Xiaodong Zhang. Understanding Intrinsic Characteristics and System Implications of Flash Memory Based Solid State Drives. In *SIGMETRICS'09: Proceedings of the 11th International Joint Conference on Measurement and Modeling of Computer Systems*, pages 181–192, Seattle, WA, USA, June 2009. ACM.

[21] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. RAID: High-performance, Reliable Secondary Storage. *ACM Computer Survey*, 26(2):145–185, June 1994.

[22] Dennis Colarelli and Dirk Grunwald. Massive Arrays of Idle Disks for Storage Archives. In *Supercomputing'02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, pages 1–11, Baltimore, MD, USA, November 2002. IEEE Computer Society Press.

[23] Peter Corbett, Bob English, Atul Goel, Tomislav Grcanac, Steven Kleiman, James Leong, and Sunitha Sankar. Row-Diagonal Parity for Double Disk Failure Correction. In *FAST'04: Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, pages 1–14, San Francisco, CA, USA, March 2004. USENIX Association.

[24] Microsoft Corporation. Microsoft Extensible Firmware Initiative FAT32 File System Specification, December 2000. Available at: `http://www.microsoft.com/whdc/system/platform/firmware/fatgen.mspx`. Last accessed on October 2011.

[25] Microsoft Corporation. Detailed Explanation of FAT Boot Sector, December 2003. Available at: `http://support.microsoft.com/kb/140418`. Last accessed on October 2011.

[26] Microsoft Corporation. How Basic Disks and Volumes Work, March 2003. Available at: `http://technet.microsoft.com/en-us/library/cc739412.aspx`. Last accessed on October 2011.

[27] Microsoft Corporation. Overview of FAT, HPFS and NTFS File Systems, May 2007. Available at: `http://support.microsoft.com/kb/100108`. Last accessed on October 2011.

[28] Microsoft Corporation. Windows SteadyState 2.5 Handbook, February 2008. Available at: `http://www.microsoft.com/download/en/details.aspx?id=4310`. Last accessed on October 2011.

[29] Koustuv Dasgupta, Sugata Ghosal, Rohit Jain, Upendra Sharma, and Akshat Verma. QoSMig: Adaptive Rate-Controlled Migration of Bulk Data in Storage Systems. In *ICDE'05: Proceedings of the 21st International Conference on Data Engineering*, pages 816–827, Tokyo, Japan, April 2005. IEEE.

[30] Giel de Nijs, Ard Biesheuvel, Ad Denissen, and Niek Lambert. The Effects of Filesystem Fragmentation. In *Proceedings of the 2006 Linux Symposium*, pages 193–208, Ottawa, Canada, July 2006.

[31] John R. Douceur and William J. Bolosky. A Large-scale Study of File-system Contents. In *SIGMETRICS'99: Proceedings of the 1999 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 59–70, Atlanta, GA, USA, May 1999. ACM.

[32] Ian Dowse and David Malone. Recent Filesystem Optimisations in FreeBSD. In *FREENIX'02: Proceedings of the USENIX Annual Tech-*

nical Conference, Freenix Track, pages 245–258, Monterey, CA, USA, June 2002. USENIX Association.

[33] The Linux Foundation. A Conversation with Chris Mason on BTRfs: the next generation file system for Linux, June 2009. Available at: `http://www.linuxfoundation.org/news-media/blogs/browse/2009/06/`. Last accessed on February 2012.

[34] Felipe Franciosi and William J. Knottenbelt. Towards a QoS-aware Virtualised Storage System. In *UKPEW'09: Proceedings of the 2009 UK Performance Evaluation Workshop*, pages 56–67, Leeds, UK, July 2009. University of Leeds.

[35] Felipe Franciosi and William J. Knottenbelt. Data Allocation Strategies for the Management of Quality of Service in Virtualised Storage Systems. In *MSST'11: Proceedings of the 27th IEEE Symposium on Mass Storage Systems and Technologies*, pages 1–6, Denver, CO, USA, May 2011. IEEE.

[36] Bezalel Gavish and Olivia R. Liu Sheng. Dynamic File Migration in Distributed Computer Systems. *Communications of the ACM*, 33(2):177–189, February 1990.

[37] Shahram Ghandeharizadeh, Douglas J. Ierardi, Dongho Kim, and Roger Zimmermann. Placement of Data in Multi-Zone Disk Drives. In *BalticDB&IS '96:: Proceedings of the 2nd International Baltic Workshop on Databases and Information Systems*, Tallinn, Estonia, June 1996.

[38] Salil Gokhale, Nitin Agrawal, Sean Noonan, and Cristian Ungureanu. KVZone and the Search for a Write-optimized Key-value Store. In *HOTSTORAGE'10: Proceedings of the 2nd USENIX Conference on Hot Topics in Storage and File Systems*, pages 6–11, Boston, MA, USA, June 2010. USENIX Association.

[39] Joseph Hall, Jason Hartline, Anna R. Karlin, Jared Saia, and John Wilkes. On Algorithms for Efficient Data Migration. In *SODA'01: Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 620–629, Washington, DC, USA, January 2001. Society for Industrial and Applied Mathematics.

[40] Craig K. Harmer and Rebanta Mitra. Reverse Pathname Lookup by inode Identifier, July 2010. US Patent No. 7752226.

[41] Peter G. Harrison, Soraya Zertal, and Naresh Patel. Response Time Distribution of Flash Memory Accesses. *Performance Evaluation*, 67(4):248–259, April 2010.

[42] Windsor W. Hsu, Alan Jay Smith, and Honesty C. Young. The Automatic Improvement of Locality in Storage Systems. *ACM Transactions on Computer Systems*, 23(4):424–473, November 2005.

[43] Bert Hubert. Naive But Spectacular ext3 HTREE+Orlov Benchmark, November 2002. Available at: `http://lwn.net/Articles/14631/`. Last accessed on October 2011.

[44] IBM. Journaled File System Technology for Linux, April 2011. Available at: `http://jfs.sourceforge.net/`. Accessed on October 2011.

[45] MiniMicroMart Inc. Advertisement on 18 MB North Star Hard Disk HD-18 for U$ 4 199. *Creative Computing Magazine*, page 5, July 1980.

[46] R-Tools Technology Inc. R-Studio, September 2011. Available at: `http://www.r-tt.com/`. Last accessed on October 2011.

[47] Internetnews.com. A Better Filesystem for Linux?, October 2008. Available at: `http://www.internetnews.com/dev-news/article.php/3781676/`. Last accessed on February 2012.

[48] Michael J. Karels and Marshall Kirk McKusick. Toward a Compatible Filesystem Interface. In *EUUG'86: Proceedings of the 1986 European Unix User's Group*, Buntington, England, UK, September 1986.

[49] Seon Ho Kim, Hong Zhu, and Roger Zimmermann. Zoned-RAID. *ACM Transactions on Storage*, 3(1):1–17, March 2007.

[50] Steve Kleiman. Trends in Managing Data at the Petabyte Scale. In *Invited Talk in FAST'07: 5th USENIX Conference on File and Storage Technologies*, San Francisco, CA, USA, February 2007.

[51] William J. Knottenbelt, Peter G. Harrison, and Soraya Zertal. Intelligent Performance Optimisation of Virtualised Data Storage Systems (iPODS), October 2007. Available at: `http://gow.epsrc.ac.`

uk/NGBOViewGrant.aspx?GrantRef=EP/F010192/1. Last accessed on October 2011.

[52] Matthew Komorowski. A History of Storage Cost, July 2009. Available at: http://www.mkomo.com/cost-per-gigabyte. Last accessed on October 2011.

[53] Abigail Lebrecht, Nicholas J. Dingle, Peter G. Harrison, William J. Knottenbelt, and Soraya Zertal. Using Bulk Arrivals to Model I/O Request Response Time Distributions in Zoned Disks and RAID Systems. In *VALUETOOLS'09: Proceedings of the 4th International Conference on Performance Evaluation Methodologies and Tools*, Pisa, Italy, October 2009.

[54] Abigail Lebrecht, Nicholas J. Dingle, and William J. Knottenbelt. A Response Time Distribution Model for Zoned RAID. In *ASMTA '08: Proceedings of the 15th International Conference on Analytical and Stochastic Modelling Techniques and Applications*, Lecture Notes in Computer Science, pages 144–157, Nicosia, Cyprus, June 2008.

[55] Abigail Lebrecht, Nicholas J. Dingle, and William J. Knottenbelt. Modelling and Validation of Response Times in Zoned RAID. In *MASCOTS'08: Proceedings of the 16th Annual Meeting of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 1–10, Baltimore, MD, USA, September 2008. IEEE.

[56] Abigail Lebrecht, Nicholas J. Dingle, and William J. Knottenbelt. Validation of Large Zoned RAID Systems. In *UKPEW'08: Proceedings of the 24th UK Performance Engineering Workshop*, pages 246–261, London, UK, July 2008.

[57] Abigail Lebrecht, Nicholas J. Dingle, and William J. Knottenbelt. A Performance Model of Zoned Disk Drives with I/O Request Reordering. In *QEST'09: Proceedings of the 6th International Conference on Quantitative Evaluation of Systems*, pages 97–106, Budapest, Hungary, September 2009.

[58] Abigail Lebrecht, Nicholas J. Dingle, and William J. Knottenbelt. Modelling Zoned RAID Systems using Fork-Join Queueing Simula-

tion. In *EPEW'09: Proceedings of the 6th European Performance Engineering Workshop*, pages 16–29, London, UK, July 2009. Springer.

[59] Abigail Lebrecht, Nicholas J. Dingle, and William J. Knottenbelt. Analytical and Simulation Modelling of Zoned RAID Systems. *The Computer Journal*, 54(5):691–707, May 2011.

[60] A. Leung, I. Adams, and E.L. Miller. Magellan: A Searchable Metadata Architecture for Large-scale File Systems, November 2009. Technical Report UCSC-SSRC-09-07.

[61] Andrew W. Leung. Organizing, Indexing, and Searching Large-Scale File Systems, December 2009. Technical Report UCSC-SSRC-09-09.

[62] A. J. Lewis. LVM HowTo, November 2006. Linux Documentation Project. Available at: `http://www.tldp.org/HOWTO/LVM-HOWTO`. Last accessed on October 2011.

[63] Linus's Kernel Tree. ext4: Rename ext4dev to ext4, October 2008. Available at: `http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;h=03010a3350301baac2154fa66de925ae2981b7e3`. Last accessed on October 2011.

[64] Chris Lueth. WORM Storage on Magnetic Disks Using SnapLock Compliance™ and SnapLock Enterprise™, September 2003. Technical Report 3263. Network Applicance, Inc. Available at: `http://web.mit.edu/dean/Research/TR3263-2.pdf`. Last accessed on October 2011.

[65] Christopher R. Lumb, Arif Merchant, and Guillermo A. Alvarez. Façade: Virtual Storage Devices with Performance Guarantees. In *FAST'03: Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pages 131–144, San Francisco, CA, USA, March 2003. USENIX Association.

[66] Wayne Marshall. Boot with GRUB. *Linux Journal*, 2001(85es), May 2001.

[67] Avantika Mathur, Mingming Cao, and Suparna Bhattacharya. The New ext4 Filesystem: Current Status and Future Plans. In *Proceedings of the 2007 Linux Symposium*, pages 21–34, Ottawa, Canada, June 2007.

[68] Makoto Matsumoto and Takuji Nishimura. Mersenne Twister: a 623-dimensionally Equidistributed Uniform Pseudo-random Number Generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, January 1998.

[69] Margaret H. McFarland. SEC Interpretation: Electronic Storage of Broker-Dealer Records, May 2003. Available at: `http://www.sec.gov/rules/interp/34-47806.htm`. Last accessed on October 2011.

[70] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[71] J. Menon, D. A. Pease, R. Rees, L. Duyanovich, and B. Hillsberg. IBM Storage Tank - A Heterogeneous Scalable SAN File System. *IBM Systems Journal*, 42(2):250–267, 2003.

[72] Ganesh Narayan and K. Gopinath. *iSAN - An Intelligent Storage Area Network Architecture*, volume 3296 of *Lecture Notes in Computer Science*, pages 262–273. Springer Berlin / Heidelberg, 2005.

[73] Evi Nemeth, Garth Snyder, Trent R. Hein, and Ben Whaley. *UNIX System Administration Handbook*. Prentice-Hall, Inc., 4th edition, July 2010.

[74] Grigoriy Orlov. Directory Allocation Algorithm for FFS, December 2000. Available at: `http://web.archive.org/web/20080131082712/http://www.ptci.ru/gluk/dirpref/old/dirpref.html`. Last accessed on October 2011.

[75] N. Park and D.J. Lilja. Characterizing Datasets for Data Deduplication in Backup Applications. In *IISWC'10: Proceedings of the 2010 IEEE International Symposium on Workload Characterization*, pages 1–10, Atlanta, GA, USA, December 2010. IEEE.

[76] David A. Patterson, Gareth Gibson, and Randy H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *SIGMOD'88: Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, pages 109–116, Chicago, IL, USA, June 1988. ACM.

[77] James S. Plank. A Tutorial on Reed-Solomon Coding for Fault-Tolerance in RAID-like Systems. *Software Practice and Experience*, 27(9):995–1012, September 1997.

[78] Red Hat^{TM}, Inc. Red Hat Enterprise Linux Deployment Guide. Chapter 9.4: Manual LVM Partitioning, January 2007. Available at: `http://www.centos.org/docs/5/html/Deployment_Guide-en-US/s1-lvm-diskdruid-manual.html`. Last accessed on October 2011.

[79] Ohad Rodeh. B-Trees, Ahadowing, and Clones. *ACM Transactions on Storage*, 3(4):2:1–2:27, February 2008.

[80] Chris Ruemmler and John Wilkes. Disk Shuffling, October 1991. HP Laboratories. Technical Report HPL-91-156. Available at: `http://www.e-wilkes.com/john/papers/HPL-91-156.pdf`. Last accessed on February 2012.

[81] Chris Ruemmler and John Wilkes. An Introduction to Disk Drive Modeling. *Computer*, 27(3):17–28, March 1994.

[82] J. Rydning and D. Reinsel. Worldwide hard disk drive 2009–2012 forecast: Navigating the transitions for enterprise applications, February 2009. IDC Market Analysis, Document 216394.

[83] Takashi Sato. ext4 Online Defragmentation. In *Proceedings of the 2007 Linux Symposium*, pages 179–186, Ottawa, Canada, July 2007.

[84] Olivia R. Liu Sheng. Analysis of Optimal File Migration Policies in Distributed Computer Systems. *Management Science*, 38(4):459–482, April 1992.

[85] The Starman. An Examination of the Standard MBR, August 2004. Available at: `http://thestarman.pcministry.com/asm/mbr/STDMBR.htm`. Last accessed on October 2011.

[86] The Starman. All the Details of Many Versions of Both the MBR and OS Boot Records, March 2011. Available at: `http://thestarman.pcministry.com/asm/mbr/`. Last accessed on October 2011.

[87] Paul J. Stoffregen and Robin Coon. Understanding FAT32 Filesystems, February 2005. Available at: `http://www.pjrc.com/tech/8051/ide/fat32.html`. Last accessed on October 2011.

[88] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the XFS File System. In *ATC'96: Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference*, pages 1–14, San Diego, CA, USA, January 1996. USENIX Association.

[89] Theodore Ts'o. Orlov block allocator for ext3, November 2002. Available at: `http://lwn.net/Articles/14447/`. Last accessed on October 2011.

[90] Theodore Ts'o. e2fsprogs: Ext2/3/4 Filesystem Utilities, May 2010. Available at: `http://e2fsprogs.sourceforge.net/`. Last accessed on October 2011.

[91] Stephen C. Tweedie. Journaling the Linux ext2fs Filesystem. In *LinuxExpo'98: Proceedings of The 4th Annual Linux Expo*, Durham, North Carolina, USA, May 1998.

[92] Hugh Ujhazy. Designing Storage Tiers, June 2005. Application Optimized Storage Solutions. White Paper.

[93] Sandeep Uttamchandani, Li Yin, Guillermo A. Alvarez, John Palmer, and Gul Agha. CHAMELEON: A Self-Evolving, Fully-Adaptative Resource Arbitrator for Storage Systems. In *ATC'05: Proceedings of the USENIX Annual Technical Conference*, pages 75–88, Anaheim, CA, USA, April 2005. USENIX Association.

[94] Aneesh Kumar K. V., Mingming Cao, Jose R. Santos, and Andreas Dilger. Ext4 Block and inode Allocator Improvements. In *Proceedings of the 2008 Linux Symposium*, pages 263–274, Ottawa, Canada, June 2008.

[95] Akshat Verma, Upendra Sharma, Rohit Jain, and Koustuv Dasgupta. Compass: Cost of Migration-aware Placement in Storage Systems. In *IM'07: Proceedings of the 10th IFIP/IEEE International Symposium on Integrated Network Management*, pages 50–59, Munich, Germany, May 2007. IEEE.

[96] Paul Vongsathorn and Scott D. Carson. A System for Adaptive Disk Rearrangement. *Software: Practice and Experience*, 20(3):225–242, March 1990.

[97] Francis Wan, Nicholas J. Dingle, William J. Knottenbelt, and Abigail Lebrecht. Simulation And Modelling Of RAID 0 System Performance. In *ESM'08: Proceedings of the 22nd Annual European Simulation and Modelling Conference*, pages 145–149, Le Havre, France, October 2008. Eurosis.

[98] Scott Watanabe. *Solaris 10 ZFS Essentials.* Prentice Hall Press, 1st edition, January 2010.

[99] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The HP AutoRAID Hierarchical Storage System. *ACM Transactions on Computer Systems*, 14(1):108–136, 1996.

[100] Soraya Zertal and Peter G. Harrison. Investigating Flash Memory Wear Levelling and Execution Modes. In *SPECTS'09: Proceedings of the 12th International Conference on Symposium on Performance Evaluation of Computer & Telecommunication Systems*, pages 81–88, Istanbul, Turkey, July 2009. IEEE.

A. Additional Experiments

A.1. Using the `ext3ipods` inode Allocator with a Reduced Filesystem Population

This section provides additional results regarding the experiments conducted with the `ext3ipods` QoS-aware inode allocator. Based on the results observed in Section 5.4.2, a conjecture was elaborated regarding the allocation of underprovisioned areas. It states that an even amount of requests are made with the QoS combinations at hand, but there is no corresponding storage space available per QoS combination to meet such requests.

To validate the conjecture, the experiment was rerun with a modified parameter set in Impressions. The new configuration requests a 2.0 GB population to the instead of the original 4.0 GB. The same evaluation mechanisms are then applied to the resulting filesystem in order to assess the results.

Table A.1 illustrate the block group allocation scores for the new test case. It is possible to immediately visualise the score of the third tier dropping to zero. The score of the second tier, however, increased considerably and further analysis is required to determine the reason why.

Next, Figures A.1, A.2 and A.3 displays the allocation bitmaps for the reliability, read and write performance respectively. Observing these results, it is possible to visualise see the original underprovisioned area disappears, suggesting that the conjecture was correct.

The only areas that are not perfectly matched appear in the second tier. Analysing these inodes individually, we observed that they actually request a QoS combination of low reliability and low read and write performance. While Impressions was not configured to issue allocation requests with this QoS combination, the root directory of the filesystem was not set with any QoS attributes. In our implementation, this is interpreted as low requirements for all attributes. Therefore files created on this top level directory will inherit such attributes accordingly.

Tier	Block Group Index	Block Group Score	Accumulated Tier Score
RAID5	1	0	0
	2	0	
	3	0	
	4	0	
	5	0	
	6	0	
	7	0	
	8	0	
	9	0	
	10	0	
	11	0	
	12	0	
RAID10	13	4014	55 650
	14	16 096	
	15	18 498	
	16	5 676	
	17	6 440	
	18	0	
	19	4 926	
	20	0	
RAID0	21	0	0
	22	0	
	23	0	
	24	0	
	25	0	
	26	0	
	27	0	
	28	0	
	29	0	
	30	0	
	31	0	
	32	0	
	33	0	
	34	0	
	35	0	
	36	0	

Table A.1.: Block group evaluation scores whilst using the QoS-aware inode allocator with reduced filesystem population.

Figure A.1.: QoS-aware inode allocator bitmap for the reliability QoS attribute when using a reduced filesystem population.

Figure A.2.: QoS-aware inode allocator bitmap for the read performance QoS attribute when using a reduced filesystem population.

Figure A.3.: QoS-aware datablock inode bitmap for the write performance QoS attribute when using a reduced filesystem population.

Finally, we evaluate the datablock distribution according to the provisioning match. Table A.2 lists the results, showing in detail that 98.34% of the allocated datablocks were perfectly matched to a corresponding block group. It also quantifies the small amount of overprovisioning to 1.66%.

	Very Underprov.	Underprov.	Perfectly Matched	Overprov.	Very Overprovisioned
Reliability	0	0	530 776	0	27 825
Read Performance	0	0	558 601	0	0
Write Performance	0	0	558 601	0	0
Total Count	0	0	1 647 978	0	27 825
Average	0%	0%	98.34%	0%	1.66%

Table A.2.: Datablock count under each provisioning category for the QoS-aware inode allocator when analysing a filesystem with reduced population.

While the results listed in this section do not constitute a formal proof to explain the underprovisioning observed in Section 5.4.2, they serve to strengthen the presented conjecture. Also, these results better illustrate the good performance of the QoS-aware allocators, considering a smaller population of the filesystem and therefore enough space for the allocator to work.

A.2. Using the `ext3ipods` Datablock Allocator with Reduced Filesystem Population

This section experiments with the `ext3ipods` QoS-aware datablock allocator using an Impressions modified parameters set. The modifications will configure Impressions to occupy 2.0 GB of the filesystem instead of 4.0 GB as performed in Section 5.4.3. The objective of this additional experiment is to observe which block groups are selected first during the allocation process.

To achieve this objective, the only evaluation required is the production of the allocation bitmaps, as the relative scores produced by the other two evaluation mechanisms are not relevant. Figures A.4, A.5 and A.6 present

the reliability, read and write performance allocation bitmaps.

Figure A.4.: QoS-aware datablock allocator bitmap for the reliability QoS attribute on a reduced population filesystem.

Considering the less populated filesystem, it is possible to observe in practice the behaviour described in the `ext3ipods` QoS-aware datablock allocation algorithm. The implementation, which follows the algorithm, scans all block groups linearly. Upon evaluating the QoS score for a particular datablock and block group, it uses the insertion sort algorithm to include the block group in the priority queue.

Figure A.5.: QoS-aware datablock allocator bitmap for the read performance QoS attribute on a reduced population filesystem.

Figure A.6.: QoS-aware datablock allocator bitmap for the write performance QoS attribute on a reduced population filesystem.

Due to the priority queue structure, which has the last evaluated block group at its beginning, the selection of block groups will actually take place from the end of the filesystem towards the start. The datablock allocation within that block group, however, will commence on the first available addresses and move towards the end.

Observing the allocation bitmaps presented in this section, this behaviour is noticeable. To recognise it, it suffices to note that each tier has the last block groups allocated and the first ones are still available. Furthermore, the last block group to where datablocks were placed (i.e. the first populated block group in each tier) will be occupied from the first addresses towards the last.