Towards a QoS-aware Virtualised Storage System

Felipe Franciosi and William Knottenbelt*

Abstract

Every organisation depends critically on reliable high-performance storage. Driven by the high costs of maintaining and managing multiple local storage systems, there is a trend towards virtualised multi-tier storage infrastructures. The main limitation of such centralised solutions is their inability to guarantee application-level Quality of Service (QoS) without extensive and ongoing human intervention. This intervention is necessary since delivered QoS can vary extensively both across and within storage tiers, and also depends on the access profile of the data.

This paper presents the first steps towards the concrete realisation of a self-managing virtualised storage system which automatically allocates and migrates data throughout its lifecycle guided by user-provided QoS hints. Specifically, we use the Logical Volume Manager (LVM) to create a virtualised multi-tier storage infrastructure with variable performance and reliability profiles. On to that, we place an enhanced (but backwards-compatible) Linux Extended 3 Filesystem which we call **ext3ipods** and which supports QoS metadata. We describe the kernel modifications necessary to quantify the QoS provided by a given data layout, thus enabling the subsequent development of intelligent data placement and migration algorithms.

1 Introduction

Regardless of the nature of any contemporary organisation, information is invariably considered to be one of its most valuable assets. In practical terms, this translates into the need for reliable, scalable and high-performance storage infrastructures. The traditional approach to meeting these requirements is to allocate separate storage solutions on a per-application basis. However, such configurations not only incur a high management overhead (since system administrators need to constantly monitor and configure many different systems), but they also suffer from serious inefficiencies due to the overprovisioning of resources such as storage capacity and power.

In an attempt to increase efficiency and reduce the total cost of ownership, recent years have seen the emergence of a trend towards virtualised multi-tier storage infrastructures [14]. These can be described as centralised storage systems organised into tiers, each of which is characterised by a set of attributes

*Department of Computing, Imperial College London, 180 Queen's Gate, South Kensington, SW7 2BZ, United Kingdom {ozzy,wjk}@doc.ic.ac.uk

such as cost per gigabyte and power consumption. Each tier is capable of meeting different Quality of Service (QoS) requirements with respect to data access, such as performance, reliability and space efficiency. In the absence of an intelligent storage fabric, the QoS requirements for each application and the attributes provided by the tiers must be manually matched by storage managers and data allocated accordingly, as illustrated in Figure 1.

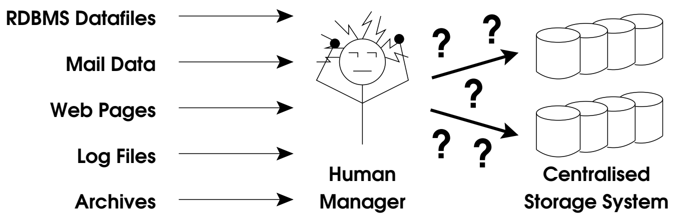

Figure 1: Human storage manager matching application QoS requirements in a centralised multi-tiered storage system.

While this drastically reduces the cost of managing several local storage systems, maintaining a centralised multi-tiered system is still a complex task. This is due not only to the very different QoS requirements imposed by each application, but also to QoS variation both *across* and *within* tiers. To see this, consider the example of a 2.5 TB logical volume comprised of two distinct RAID [13] arrays, each of which is made up of four 500 GB zoned SCSI disks[1]. The first array is RAID5, providing greater performance and space, but smaller reliability. The second array is RAID01, providing less space and performance, but greater reliability.

Sequentially reading 100 MB blocks of data across this logical volume produces the data presented in Figure 2. While a difference in performance between the two arrays is noticeable, it can be seen that the use of zoned disks causes performance variations within tiers, forming what is termed a Zoned-RAID [10] and complicating data allocation [9]. Needless to say, this performance profile would likely differ under other access patterns, such as random I/O requests of varying types and sizes. The latter is a particularly important concern when working with Solid State Drives (SSDs). Another detail that should be taken into account is the way data access evolves over its lifecycle; as an example, a news story that is currently highly accessed might be much less popular in the future, and may therefore not require high-performance storage.

This paper presents the first steps toward the practical realisation of an intelligent storage fabric that autonomously allocates and migrates data in a virtualised multi-tiered storage system using user-provided QoS hints. We aim to improve on the mechanisms for fabric intelligence in current systems which employ simple inter-tier migration policies based on access frequency, and which are centred on capacity utilisation and failure recovery.

The remainder of this paper is organised as follows. Section 2 compares our ideas to existing solutions and discusses the importance of matching data placement to QoS requirements. Section 3 shows how we created a multi-tier filesystem testbed, and discusses our rationale for placement of our intelligent fabric at the filesystem level. Section 4 presents our first steps towards the

[1]More details about this experimental setup are discussed on Section 3.

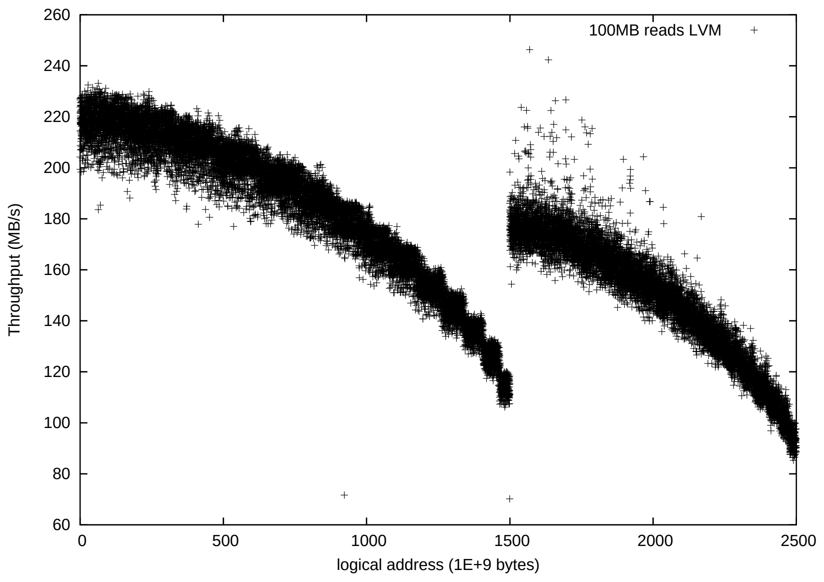

Figure 2: Sequential 100 MB reads on a multi-tier 2.5 TB logical volume.

implementation of a Linux filesystem that supports QoS awareness, along with some auxiliary tools that we have developed. Section 5 concludes.

2 QoS Requirements in Storage Systems

Current solutions for the autonomous management of virtualised multi-tier storage systems attempt to optimise *average* (or overall) system performance by allocating, migrating and reconfiguring data layouts on the basis of data access profiles [1, 2, 15]. These profiles usually consider the type of I/O (read or write), the size of the requests and access frequencies, and sometimes also pay attention to the cost involved in the reconfiguration process [4, 16]. However, it is impossible to deduce the business-level importance of rapid access to data solely on the basis of access profile. For example, a critical – but relatively infrequent – real-time transaction (e.g. an algorithmic trade) may depend on rapid access to tables in an RDBMS. Ignorant of this business need, current solutions would likely place this data on a lower performance tier.

There are other QoS requirements that are not supported by current solutions, but which may be important from a business perspective:

- **Reliability:** Access profiles are not sufficient to determine the reliability requirements of data. As an example, temporary files, which may not require redundancy, could be accessed in a similar manner to RDBMS data files, which usually require the highest reliability available in the system. Providing high reliability for the whole storage pool is an obvious waste of resources. The reliability requirements of data may also evolve over its lifecycle; for example, compliance logs may need to be kept on highly reliable storage for a set amount of time, after which the requirement for reliability lapses.

- **Space efficiency:** The efficient use of disk capacity is an important concern, although it must frequently be traded-off against performance and reliability. For example, RAID1 typically provides good reliability and read performance at the cost of low space efficiency. On the other hand, RAID0 provides good performance and space efficiency at the cost of low reliability. Software-based solutions such as the use of lossless or lossy compression algorithms may also be considered where performance is not a primary concern.

- **File growth likelihood:** Intelligent data placement can be greatly facilitated by knowing the likelihood with which files stored in the system are likely to grow. If the system is aware that the file is a static video file that should not grow over time, another piece of data can be allocated in a physical area immediately after its end. Usually, the operating system would try to preallocate some space for that file to grow, in order to avoid fragmentation [5]. Analogously, log files could have a greater space for growth preallocated, since they are very likely to grow.

- **Security sensitivity:** As discussed, many systems migrate data in order to improve overall average system performance. Data migration, however, is usually done by copying files to a new physical location on the storage system, updating pointers and abandoning the old data. This situation could lead to a security hazard scenario [7, 8], where a disk that is no longer in use – but still holds sensitive information – could be disposed of or reallocated. If the storage system were to be aware of such particularities, it could enforce data deletion upon migration.

Unlike systems that try to automatically identify what are the access profiles and optimise data layout to improve *average* performance, our work introduces a new approach. Our idea consists of allowing users to provide hints to the storage system that make it aware of a wide variety of QoS-related requirements as dictated by business needs. Figure 3 illustrates how we visualise a system that uses these hints to intelligently map data in a multi-tier storage system.

Naturally, our system will also need to be aware of the QoS delivered by each tier. This could be done either by measurement (e.g. to obtain performance profiles) or by a system administrator (e.g. with regards attributes such as reliability and power consumption). In the illustrated example, temporary files and the database should reside on high performance disks, with the latter also receiving the highest possible level of redundancy. The logs and archives, on the other hand, should be allocated to tiers that offer reliable disk space but less performance. Details on how candidate allocations can be compared and evaluated will be discussed further in Section 4.

We note that the implementation of an intelligent fabric could take place at several different levels. The following section discusses our motivation for having the QoS definitions on a filesystem level, explaining the granularities used in our work and how the multi-tier virtualised storage infrastructure was defined and created.

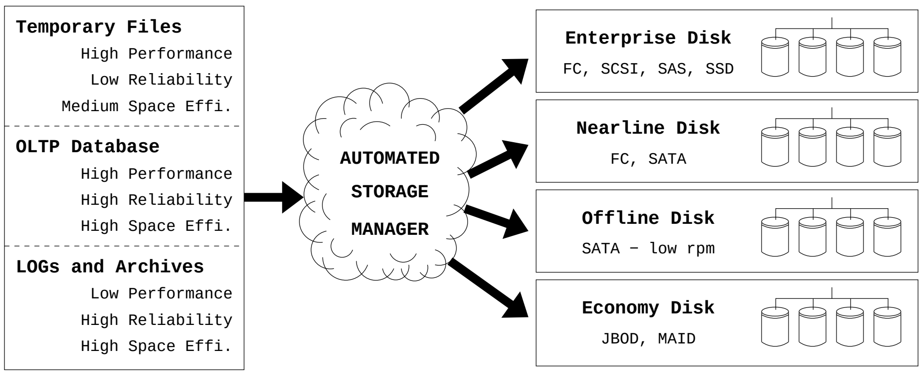

Figure 3: An intelligent fabric for mapping data in a multi-tier virtualised storage system using QoS hints.

3 Towards a QoS-aware Virtualised Filesystem

To provide an experimental framework for the problem at hand, it is first necessary to construct a multi-tiered logical volume capable of providing different levels of performance, reliability and space efficiency. Further, a mechanism is necessary to control data placement. This not only allows allocation and migration strategies to be applied, but also enables the evaluation of different data layouts.

Concretely, our experimental environment consists of eight 500 GB Seagate ST3500630NS zoned SCSI disks housed in an Infortrend A16F-G2430 RAID enclosure that is connected via a dual fibre channel interface to a Ubuntu 7.04 Linux box with 4 GB of RAM and two dual-core AMD Opteron 2218 2.5 GHz Processors. Using Linux's Software RAID (MD) [12] version 2.5.6, it was possible to create two software RAID[2] arrays of type 5 and 01 that were concatenated using Logical Volume Manager (LVM) [11] version 2.02.06. This configuration is illustrated in Figure 4 and the performance chart for 100 MB blocks sequential reads was already presented in Section 1. The point where the I/O requests are directed to the second RAID array is clearly apparent at the 1.5 TB logical address.

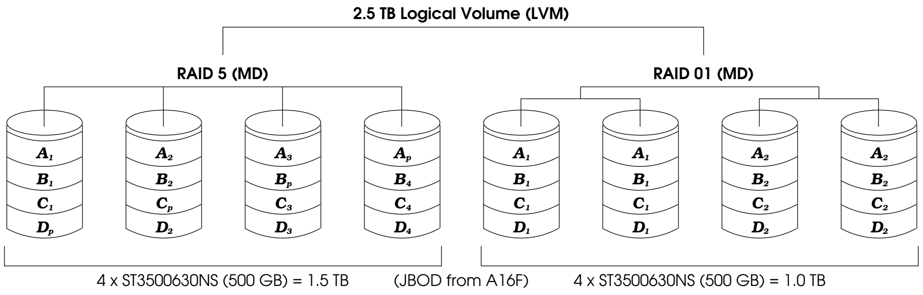

Figure 4: A 2.5 TB logical volume composed of two tiers.

[2]We note hardware RAID could have been used; however, software RAID provides similar performance while allowing for disk-level inspection and monitoring.

The next step is to decide where the intelligent fabric should be inserted into the storage system architecture. As shown in Figure 5, we shall consider that an application is the topmost level, imposing QoS requirements and accessing storage through an operating system's filesystem API. At the lowest level the storage infrastructure provides the capability to deliver various levels of QoS.

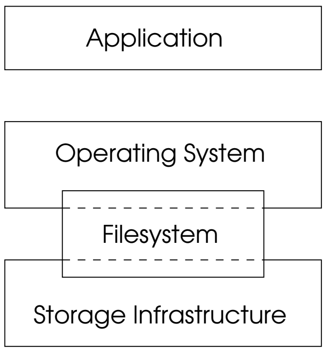

Figure 5: Levels in a storage system architecture.

Ideally a solution should allow existing applications to run without modification; this precludes insertion into the application layer. Similarly, modification of the storage infrastructure layer is problematic since raw storage device interfaces do not currently support QoS flags; nor are they likely to do so for the foreseeable future.

The operating system layer, and specifically the filesystem, provides the natural home for our intelligent fabric through metadata attached to files and directories. The latter can be used to inform intelligent kernel datablock allocation without the need to modify either applications or storage devices. It can also help an OS to evaluate the state of current data allocation, calculate improved scenarios and migrate data to meet them.

To achieve this, we must provide mechanisms to add, modify or remove file and directory hints that indicate QoS requirements for the data at hand. Naturally, a convenient place to keep this metadata is in the filesystem itself. In the same way, the filesystem should be enhanced with a means to store metadata regarding the QoS profile of its underlying storage. This not only enables persistence of such information, but also permits easy access to this data by kernel datablock allocation algorithms.

To this end, the next section analyses the structure of the Linux Extended 3 Filesystem and explains how we have enhanced it to hold such attributes while keeping the new filesystem compatible with the original one.

4 The Linux Extended 3 iPODS Filesystem

Named after the project[3] that made this research possible, the Linux Extended 3 iPODS Filesystem (ext3ipods) is our proposed extension to the Linux Extended 3 Filesystem (ext3fs). So far this extension has taken place in three main steps.

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

Firstly, we have modified the existing ext3fs disk structure, defining how to store QoS metadata in a way that maintains backwards compatibility. This has been established as a prerequisite, since it is useful to be able to mount existing filesystems with our new kernel implementation and hence evaluate QoS gains or losses more easily. Secondly, we have provided a means for both users and system administrators to update these new attributes. Finally, we have considered what new tools and strategies within the kernel are necessary to improve datablock allocation using the newly defined metadata. The next three subsections deal with each of these steps respectively.

4.1 ext3ipods Disk Structure

To define how ext3fs should be modified to store the newly specified metadata, we have initially studied what granularity should be used to define QoS requirements. Considering that applications usually use directories and files as their main storage units, the filesystem's inode has been chosen. Conveniently, this already possesses a field, named *flags*, capable of holding such metadata. This field is 32 bits long and is not currently used in its entirety, allowing us to use the free space. ext3fs also supports an inode attribute extension that could hold a greater amount of flags should this become necessary in the future.

The next modification is to define how to store the QoS characteristics provided by the virtualised storage infrastructure layer. Taking into account that storage in ext3fs is divided into block groups [3], and that current allocation algorithms use that unit to keep what it considers similar data grouped together [6], block groups are a natural choice for the granularity of such metadata. A control structure, named a block group descriptor, holds statistical information for every block group (such as the number of free inodes or blocks) and offers 96 bits of unused space in the latest implementation. This space will be used by ext3ipods not only to hold the aforementioned metadata, but also to store information on the effectiveness of current data allocation. This will be discussed further in Section 4.3.

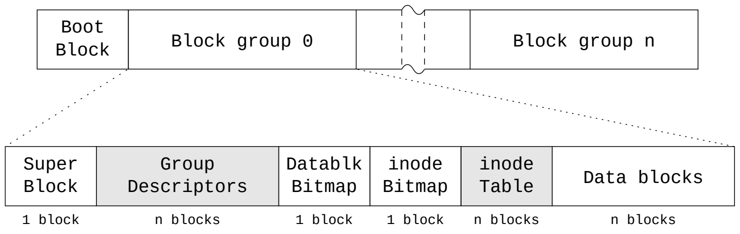

Figure 6: Underlying structure of the ext3ipods filesystem.

Figure 6 illustrates the structure of a regular ext3fs, showing how this filesystem is organised as an array of block groups, and highlighting the two parts of the disk structure that have been modified in ext3ipods. Because the extra metadata we introduce in the modified parts are stored in space that is unused in ext3fs, an ext3ipods filesystem can easily be mounted as a regular ext3fs and vice versa, maintaining compatibility.

We are currently using the extra metadata introduced into the inode flags to denote desired performance, reliability and space efficiency. Each one of them can be set on a directory or file to low, medium or high. When set on directories, new child files automatically inherit the parent directory's attributes. It is worth mentioning that when a file is created, the kernel first allocates an inode and then starts allocating datablocks as the contents start to be written. This gives no opportunity for flags to be set on the inode, as the datablock allocator will start to work immediately. By allowing new files to inherit flags from the parent directory, this problem is easily solved.

Returning to the issue of the granularity for the QoS characteristics provided by the storage layer, which has been assigned to block group units, we note that, depending on the size of each datablock, the size of the block groups in the filesystem will change. This happens because every block group holds a bitmap that is a bit vector responsible for mapping which datablocks are available. Since the bitmap is stored in exactly one datablock, the bitmap is able to address up to eight times its own size, as shown in Table 1.

Block size	Blocks addressed by bitmap	Block Group Size
1,024 B	8,192	8 MB
2,048 B	16,384	16 MB
4,096 B	32,768	32 MB

Table 1: Relation between datablock, bitmap and block group sizes.

This issue brings an extra element to the choice of an adequate block size, which is usually selected according to the expectations about the size of directories, files and the filesystem itself. As an example, if a 4 KB block size is chosen for a filesystem that will host many small files and directories, a lot of internal fragmentation should be expected, resulting in wasted space. With respect to ext3ipods storage structures with several QoS variations within a tier (as is the case in zoned RAIDs), or with many small tiers, using smaller block groups should be considered to support a sufficiently fine grain for QoS characteristics.

4.2 User Space Utilities

To enable applications and users to manage regular ext3fs filesystems, a set of user-space tools are provided by the developers. This set is named **e2fsprogs** and includes filesystem checking tools, debugging utilities and user-space programs to communicate with the kernel. Two of the user-space programs are of particular interest to this work: the `lsattr` and `chattr` inode attribute manipulation tools. Filesystem formatting tools such as `mkfs` do not require any modifications since the initial disk structure of ext3ipods is identical to that of ext3fs.

Through a set of predefined `ioctl()` calls, `lsattr` is capable of requesting inode attribute information from the kernel; it then displays this information to a user in the same fashion as the popular `ls` tool. To suit the new QoS flags defined in this work, modifications on both sides of this interface had to be made. The first one, on kernel code, was to extend the set of inode flags only for files and directories (since other types of inodes, such as devices and symbolic links are also supported). The masks controlling which flags can be retrieved by users through `ioctl()` calls were also updated. The second one, on the `lsattr` code, was to extend the tool's set of flags to match those of the kernel and to include user-friendly descriptions of the new QoS attributes.

Similarly to the listing attribute tool, modifications also had to be made to `chattr` in order to allow user-space programs to alter inode flags. In a similar way, the list of arguments accepted by `chattr` and the internal set of attributes had to be augmented to cope with the new set of QoS flags. In the kernel code, the mask for user-modifiable inode attributes was also updated.

4.3 The iPODS Filesystem Manager Kernel Module

While the above changes are enough to allow user-space programs to view and manage inode flags, further kernel modifications are necessary to make actual use of them. At this stage of the iPODS project, specific data allocation and migration strategies are yet to be defined. However, we have already implemented a set of kernel operations to evaluate the current state of an ext3ipods filesystem. These operations have been implemented as a loadable module named **ifm_lkm** and, to avoid modifying the kernel's `ioctl()` interface, are called by a user-space program through a special character device. To communicate with ifm_lkm, an application should open the device for reading and writing, write a command and then read the response.

When opening the device, ifm_lkm scans the mountpoint list for any mounted ext3ipods filesystem and reads its superblock. At this stage, we are considering that no more than one such filesystem will be mounted simultaneously. The kernel module locks itself once the character device is opened in order to prevent multiple user-space applications from issuing concurrent commands. Three operations are currently available and a fourth is under development. These are:

- `info`: When issued with the request "info", ifm_lkm will reply with the block size, the number of blocks per group and the total number of blocks in the filesystem. From these, it is possible to derive other relevant data such as size of blocks and the size of the filesystem itself. Other information such as inode usage, the number of files and directories and available space will be provided in the near future.

- `getXX`: When issued with the request "getXX", where "XX" should initially be set to zero, ifm_lkm will begin listing all the inodes in use in the filesystem and their associated metadata. This includes the QoS flags defined for each inode and the position of every datablock allocated to it. Because datablocks for an inode can reside in a different block group from the inode itself, this listing allows one to determine if datablocks with a certain QoS definition are placed in unsuitable block groups. The reason for the "XX" parameter to exist is due to the buffer size indicated on `read()` system calls. When reading from a character device, the kernel fills the user-space buffer with the requested data and then returns. User space programs should then resume reading, by issuing repeating `read()` calls, in order to finish fetching all available data. The "XX" parameter will, in this case, be incremented internally by the kernel to indicate what is the next inode to be evaluated.

- `setXX:YY`: When issued with the request "setXX:YY", where "XX" indicates a block group and "YY" a set of QoS flags, ifm_lkm will promptly validate the block group number, fetch the descriptor for that block group and update its QoS metadata. The kernel buffer where this descriptor resides is then marked as dirty, so that the Linux kernel memory manager saves it back to the filesystem in a suitable manner.

- `evalXX`: The last command, “evalXX”, where “XX” indicates a block group number and should be initially set to zero, is still under development. Its purpose is to assess the quality of the current data layout with respect to the desired QoS indicated in inode flags, and the actually delivered QoS as indicated by the metadata stored within block groups. When first called it will allocate two temporary arrays within the kernel; the first one caches the QoS flags for each block group and the second one stores the current QoS provided by each block group. These arrays are needed due to the aforementioned fact that datablocks allocated to a certain inode may reside in different block groups. After the evaluation is completed, an indication of the quality of the match between desired and achieved QoS of the underlying datablocks within a block group is cached within the block group descriptor. The “XX” parameter can then be updated in the same fashion as for the “getXX” operation.

As described above, the computation of the QoS match for a block group is obtained by scanning all inodes in the filesystem. For every inode, the QoS flags of the array of datablocks in use is compared to the QoS attributes of the block group it resides in according to Equation 1. The result of this equation are accumulated in a block group quality counter which, eventually, indicates the quality of QoS matching.

$$\sum_{i \in (qnt)} \Delta_i \times m_i \tag{1}$$

In the equation, the set *qnt* stands for all quantitative QoS attributes defined in the system (e.g. performance, reliability). Δ_i indicates the difference between the level of QoS requested by an inode and the one provided by a block group. m_i indicates a multiplier that could take place in the equation to prioritise some attributes over others. As an example, performance could be more important than reliability in some cases, and vice versa.

5 Conclusions

The trend towards virtualised multi-tier storage infrastructures continues unabated. However, since delivered QoS can vary massively both across and within storage tiers, it remains a major challenge to ensure that applications receive a QoS that is appropriate to their evolving business needs. Since manual intervention by system administrators to place and migrate data appropriately incurs very high overhead, this paper has presented our first steps towards the concrete realisation of a centralised storage system capable of automatically and intelligently placing and migrating data on the basis of QoS hints. Specifically, we have presented backwards compatible extensions to the ext3fs Linux filesystem that provide awareness of both the QoS requirements of files and directories and the QoS delivered by the underlying datablocks; this facilitates a quantitative assessment of the degree QoS match provided by a given data layout. This, in turn, will subsequently be used in the development of intelligent data placement and migration algorithms.

According to that, the next natural steps for our research is to extend the work on how to evaluate data allocation in the ext3ipods filesystem. Once we are able to evaluate and compare different allocation scenarios, it will be possible to begin the studies on migration and the involved costs. Eventually, this will lead to a complete solution for virtualised storage systems.

References

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

[2] 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, 2005.

[3] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*, chapter 18. O'Reilly, Sebastopol, CA, USA, 3rd edition, November 2005.

[4] 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.

[5] Giel de Nijs, Ard Biesheuvel, Ad Denissen, and Niek Lambert. The Effects of Filesystem Fragmentation. In *Proceedings of the Linux Symposium*, volume 1, Ottawa, Ontario, Canada, July 2006.

[6] Ian Dowse and David Malone. Recent Filesystem Optimisations in FreeBSD. In *Proceedings of the USENIX Annual Technical Conference (FREENIX Track)*, Monterey, California, USA, June 2002.

[7] Simson L. Garfinkel and Abhi Shelat. Remembrance of Data Passed: A Study of Disk Sanitization Practices. *IEEE Security & Privacy*, 1(1):17–27, 2003.

[8] Tal Garfinkel, Ben Pfaff, Jim Chow, and Mendel Rosenblum. Data Lifetime is a Systems Problem. In *EW11: Proceedings of the 11th workshop on ACM SIGOPS European Workshop*, page 10, New York, NY, USA, 2004. ACM.

[9] Shahram Ghandeharizadeh, Douglas J. Ierardi, Dongho Kim, and Roger Zimmermann. Placement of Data in Multi-Zone Disk Drives. In *Second International Baltic Workshop on Databases and Information Systems*, June 1996.

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

[11] A. J. Lewis. LVM HOWTO. Linux Documentation Project, November 2006. `http://tldp.org/HOWTO/LVM-HOWTO/`.

[12] Jakob Ostergaard and Emilio Bueso. The Software-RAID HOWTO. Linux Documentation Project, June 2004. `http://tldp.org/HOWTO/Software-RAID-HOWTO.html`.

[13] David A. Patterson, Garth 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, New York, NY, USA, 1988. ACM.

[14] Hugh Ujhazy. Designing Storage Tiers, 2005. Application Optimized Storage Solutions From Hitachi Data Systems. White Paper.

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

[16] 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.