Data Allocation Strategies for the Management of Quality of Service in Virtualised Storage Systems

Felipe Franciosi and William Knottenbelt
Department of Computing, Imperial College London,
South Kensington Campus, SW7 2AZ, United Kingdom
{ozzy,wjk}@doc.ic.ac.uk

***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.**

This paper presents our work on the design and implementation of data allocation strategies in an enhanced version of the popular Linux Extended 3 Filesystem. This enhanced filesystem features support for the specification of QoS metadata while maintaining compatibility with stock kernels. We present new inode and datablock allocation strategies which seek 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 modified the Impressions benchmarking framework to support QoS metadata. The effectiveness of the resulting data allocation in terms of QoS matching is evaluated using a special kernel module that is capable of inspecting detailed filesystem allocation 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.

I. Introduction

For 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 January 1980, from the order of US$ 200 000 per gigabyte to less than US$ 0.10 per gigabyte today. 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 800 000 petabytes. This surge in data volumes is anticipated to continue; indeed by 2020 it is expected that there will be 35 zettabytes of data to manage [1].

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 rapidly-changing and evermore-stringent 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 VSSs.

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 operations.

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 [2]–[8]. The 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 for many Linux distributions, and has a large installed user base.

There are several advantages to our realised enhancements to `ext3fs`. Firstly, specification of QoS requirements is as simple as modifying file 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, we do not support the specification of complex absolute QoS requirements such as "95% of I/O requests to file F must be accomplished within 10 ms". Instead, we provide support for combinations of simpler relative QoS goals such as "the file F should be stored in one of the most highly performant areas of the logical storage pool". Similarly we allow for system administrators to

978-1-4577-0428-4/11/$26.00 © 2011 IEEE

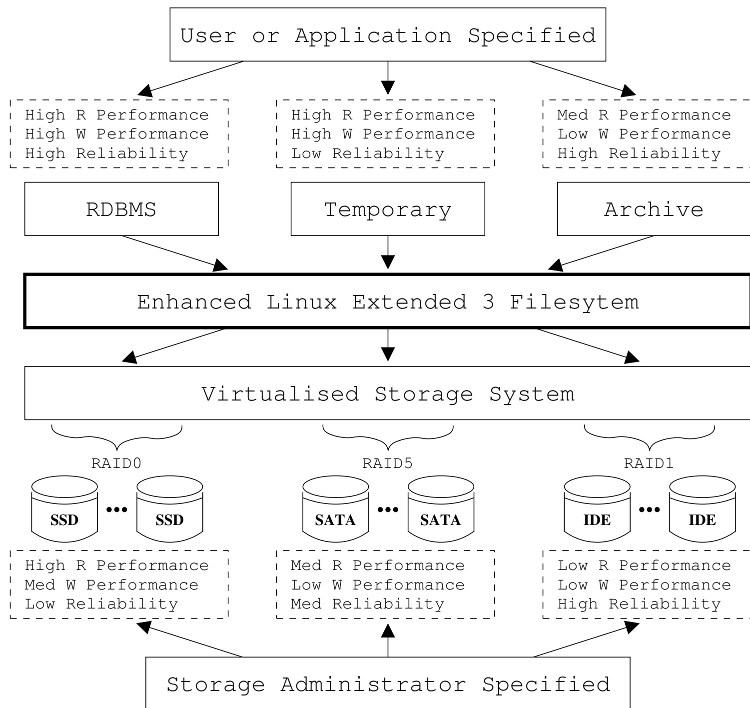

Fig. 1. QoS hints are provided by a user and matched through an enhanced filesystem fabric to a virtualised storage system profiled by a storage administrator.

easily specify the relative levels of QoS delivered by different range of block groups within the logical storage pool; this process is supported by automated performance profiling tools.

Our approach, illustrated in Figure 1, also differs from solutions that attempt to automatically deliver high levels of performance by analysing frequency and type of data accesses [2]–[4], [6], [7]. These systems not only ignore other aspects of QoS such as reliability, 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).

The focus of this paper is to show how we enhance the `ext3fs` to become QoS-aware. Firstly, we provide a mechanism to compute a score which quantifies the degree of QoS match for any given data layout. Secondly, this score is used in a placement algorithm to locate suitable block groups for new inodes and datablocks. Lastly, we introduce evaluation methods and show how our new strategies provide improved data allocation. Benchmarking of the latter is done using the Impressions [9] filesystem generation framework, which was specifically designed to create statistically accurate filesystem images and that we have modified to support QoS metadata.

The remainder of this paper is organised as follows. Section II reviews the `ext3fs` enhancements to support QoS metadata. Section III presents our strategies towards the evaluation of data placement in a populated filesystem. Section IV introduces our QoS-aware allocation algorithms. Section V presents our multi-tier storage testbed and discusses actual measurements on test cases generated with Impressions. Section VI concludes and presents our ideas for future work.

II. Enhancing `ext3fs` with QoS Support

The Linux Extended 3 iPODS Filesystem (`ext3ipods`) was conceived by the authors as a QoS-enhanced extension to the Linux Extended 3 Filesystem (`ext3fs`). Some initial work towards the practical realisation of this filesystem has been described in [10]. Specifically, it has been shown how the existing `ext3fs` filesystem layout and associated data structures can be modified in such a way as to support QoS metadata while maintaining forwards and backwards compatibility (i.e. filesystems may be mounted in either direction without the need for any conversion). This two-way compatibility not only makes deployment of the QoS enhancements into existing systems trivial but also provides an easy way to compare filesystems with and without QoS-related enhancements. An interface was also defined for both users and system administrators to update these new attributes. These features are discussed in more detail below.

A. Disk Structure

The first major design decision in considering how `ext3fs` was to be modified in order to support the QoS metadata was to chose the granularity at which QoS requirements were to be defined. Considering that applications usually use directories and files as their main storage units, the filesystem's inode was chosen for this purpose. Conveniently, inodes already possess a field named *flags*. This field is 32 bits long and currently not used in its entirety. `ext3fs` also supports an inode attribute extension that could hold a greater amount of flags should it become necessary in the future.

The extra metadata introduced into the inode flags is used to denote desired relative levels of read and write performance as well as reliability. Each one of them can be set 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 usually start to work immediately[1]. By allowing new files to inherit flags from the parent directory, this problem is easily solved without the need to modify existing application code.

The second major design decision was to choose how to store the QoS delivered by the storage infrastructure. Taking into account that `ext3fs` is divided into block groups (BGs) [11], and that allocation algorithms use that unit to keep similar data grouped together [12], BGs were a natural choice for the granularity of such metadata. A control structure, named a block group descriptor, holds information on each BG (such as the number of free inodes or blocks) and offers 96 bits of unused space. We use this space to store two types of metadata. The first is the QoS delivered at the position in the tier where that BG resides. The second is the current allocation score for that BG. Section III describes how this score is calculated.

Figure 2 illustrates the structure of a regular `ext3fs`, showing how this filesystem is organised as an array of block groups, and highlights the two parts of the disk structure that have been modified in `ext3ipods`. Because the extra metadata introduced 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.

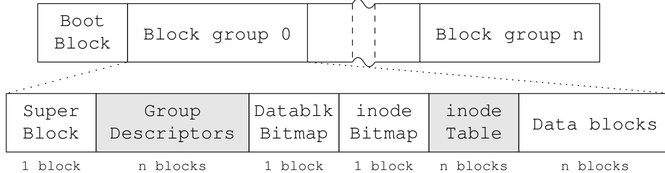

Fig. 2. Underlying structure of the `ext3fs` filesystem. The highlighted components are modified in `ext3ipods`.

B. QoS Management Interface

As discussed, 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.

To manage flags in the inodes, `e2fsprogs` [13] was extended to support the predefined metadata. This utilities set includes two tools named `chattr` and `lsattr` for setting and viewing inode attributes respectively. The `chattr` tool can be used to set desired QoS flags on files and directories in the same straightforward manner that `chmod` is used to manipulate read, write and execute flags. While applications may also invoke these tools, `e2fsprogs` also provides `libe2p`, a library that allows for the flags to be managed directly by calls made within a software.

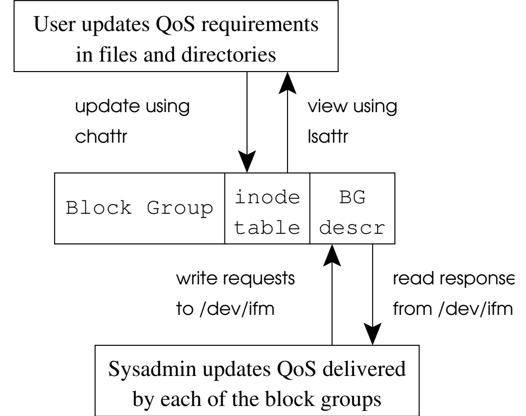

Fig. 3. QoS Management Interfaces for Users and System Administrators.

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. For this to be possible while the filesystem is mounted and running, a kernel module named iPODS Filesystem Manager (`ifm`) was developed. Communication with this module takes place through a custom character device.

While Figure 3 illustrates how users and system administrators interact with the filesystem regarding the updates of QoS attributes, `ifm` provides some further relevant resources: reporting which inodes and datablocks are in use and calculating the allocation score for a given block group. As these two functions relate directly to the evaluation of QoS match in a filesystem, they are explained further in the next section.

III. Evaluating QoS Matching

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 placement with respect to the desired and delivered QoS attributes. To do this, we compute a score for every datablock in a block group as follows:

$$\begin{aligned} &\textstyle\sum_{i\in\{attr\}} p \times \Delta_i \times m_i, \text{ where} \\ &\Delta_i = (dlv_i - req_i), \text{ and} \\ &p = \begin{cases} -|c_1| & \text{if } \Delta_i < 0 \\ |c_2| & \text{if } \Delta_i \geq 0 \end{cases} \end{aligned} \tag{1}$$

The closer to *zero* the result of Eq. 1, the better we regard the allocation for that datablock. Here $\{attr\}$ is the set of QoS attributes used (in our case, read performance, write performance and reliability). Δ_i represents the difference, with respect to QoS attribute i, between dlv_i – that is, the level of QoS delivered by the BG in which the datablock resides and req_i – that is, the level of QoS required by the inode that owns the datablock. When working only with *high*, *medium*

[1]With the exception of uncommon file creation operations such as "touching" a non-existent file, i.e. allocating the inode without writing any data.

and *low* relative QoS levels, req_i and dlv_i can be set to *two*, *one* and *zero* respectively. m_i represents a multiplier for a particular attribute; this is 1 by default but may be adjusted by an administrator to control the relative importance of QoS attributes; for example (s)he may wish to suggest that reliability should be prioritised ahead of read and write performance. Finally, p represents the provisioning factor. If Δ_i is greater than *zero*, this means that overprovisioning is taking place (i.e. the QoS delivered is better than the requested). The attitude of the system administrator to under- and overprovisioning is controlled by the values of c_1 and c_2 respectively. For our purposes, since we believe underprovisioning is worst than overprovisioning, we set $c_1 = 2$ and $c_2 = 1$.

The corresponding QoS match score for a BG is the sum of the QoS match scores of its constituent datablocks. After computing this score for every BG, `ifm` stores the result in the block group descriptor. The reason why this value is not computed by the allocator alone, during allocation and removal of datablocks, is due to the backwards compatibility of `ext3ipods`. That is, a filesystem mounted and populated by a stock `ext3fs` kernel would make this score inconsistent.

Retrieving and analysing BG scores for an entire filesystem will then provide an overall idea of the allocation quality regarding the QoS attributes. To achieve this, `ifm` generates a report of every inode in use, while also indicating their QoS requirements and a list of every datablock in use by them.

IV. Allocation Strategies

We now discuss three different allocation strategies that we made use of while conducting our study. 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.

A. Default `ext3fs` Allocator

Naturally, the default `ext3fs` inode and datablock allocators are completely unaware of any QoS metadata in the filesystem. Therefore, their algorithms are focused on keeping directories and files that are alike close to each other on the physical media, and reducing fragmentation.

To achieve this, the *inode allocator* exploits the concept of BGs. When finding a place for a new inode, the allocator will attempt to use the same BG 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 BGs 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 space in the same BG as the inode (specially when dealing with the first datablock for the inode). This is relevant to our work, 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 BG or there is a severe imbalance of inode modes in that BG, the datablock allocator will search for space elsewhere, giving preference to BGs 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.

B. QoS-aware Inode Allocator

While the original `ext3fs` inode allocator will adopt different strategies for directories or files in an attempt to spread data unlikely to be related across the filesystem, our goal is to group data that has the same QoS requirements in the set of BGs 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 III to scan for BGs capable of providing a score of *zero* (which is a perfect match between what is provided by the inode and delivered by the BG). Naturally, for a BG to be eligible, it must have free space in its inode table.

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 *zero*. This search is currently been done linearly from the first block group in the filesystem, an overhead which we have found to be acceptable in our experiments to date. However, in the interests of a more elegant approach, other methods using caching and indexing are currently under study.

In an attempt to reduce the search process (such that it is easier for a *zero* match to happen), 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 a large range of scenarios, this proves not to be ideal due to the rapid exhaustion of inode space in block groups capable of offering high QoS parameters.

C. 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.

To prevent unnecessary external fragmentation, we give preference for contiguous datablock allocation. Because the first datablocks associated with an inode are usually allocated in the same BG as the inode, they should already be in the best possible location QoS-wise. In cases where it is not possible to allocate contiguous datablocks, our new allocator generates a priority queue of candidate BGs (those that have available space) sorted according to QoS match score.

V. EXPERIMENTAL RESULTS

Our first step towards assembling a practical experimental environment was to investigate the behaviour of a real multi-tier VSS. Having at our disposal an Infortrend EonStor A16F-G2430 enclosure connected via a dual fibre channel interface to a Ubuntu 7.04 (kernel 2.6.20.3) Linux box with two dual-core AMD Opteron processors and 4GB of RAM memory, we configured three tiers and assembled them together in a single logical volume, thereby obtaining a multi-tier VSS. Each tier was composed of four 500 GB Seagate ST3500630NS zoned SCSI disks, configured according to a different RAID level in order for each tier to deliver different QoS attributes.

A. Multi-tier VSS Organisation and Profiling

The first logical addresses of our single logical volume represent the first tier, which is a 1.5 TB RAID5. Providing distributed striped parity, this array supports up to one disk failure without losing data and therefore was classified as having *medium* reliability. The next tier is a 1.0 TB RAID01, that is formed by mirrored stripes, allowing for up to two disks failure without data loss (provided they do not hold the primary and mirror copies of the same data). This tier was classified as having *high* reliability. The last tier is merely a RAID0, providing *low* reliability, but 2.0 TB of space. They were concatenated using Linux's Logical Volume Manager [14] version 2.02.06.

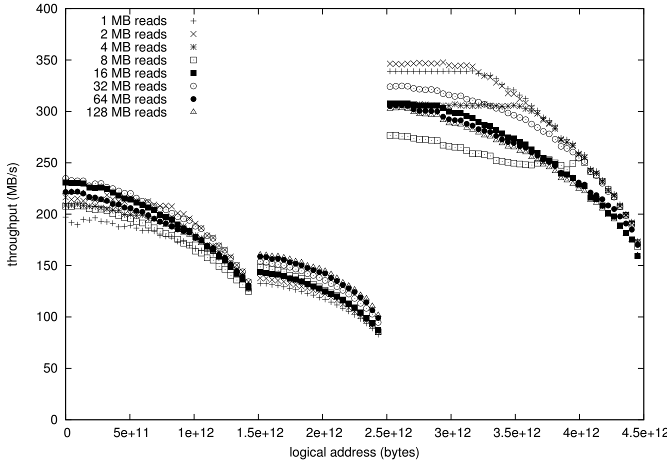

Fig. 4. Throughput of sequential reads of varying size across a VSS.

As noted earlier in this paper, QoS delivered in such configurations can vary drastically both across and within tiers. Apart from the reliability contrasts just noted, the read and write performance also differs. To show this, we have profiled our system by reading and writing different sized buffers directly to the raw device as it is seen by the operating system. Not only does this avoid the overhead induced by the filesystem itself, but it also allows for us to easily control which logical address is being profiled, giving an accurate perspective on device performance. The results are plotted in Figures 4 and 5.

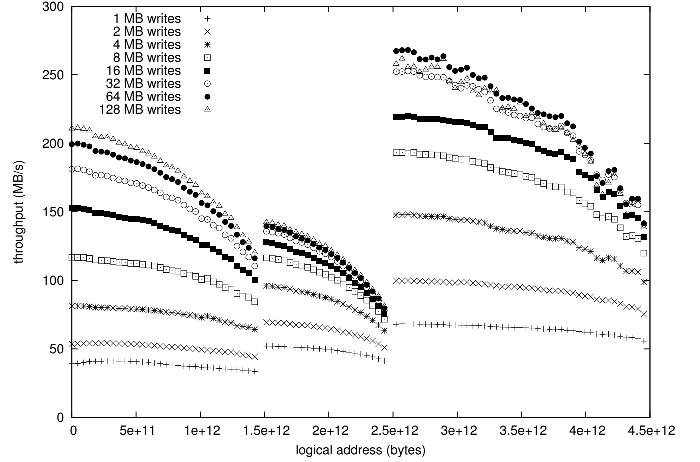

Fig. 5. Throughput of sequential writes of varying size across a VSS.

Considering our proposal focuses on relative QoS, we have classified the inferred reliability and the profiled read and write performance attributes as *low*, *medium* or *high*. This matching should be done on a per system basis; that is, whilst we consider RAID5 to be of *medium* reliability, for supporting *some* sort of disk failure without data loss, other administrators may classify RAID5 as *low* (or perhaps *high*) reliability depending on what the other tiers provide. Naturally, this may be changed at a later time if other tiers are added to the storage infrastructure. Table I shows our classification.

RAID Level	Write Performance	Read Performance	Reliability
RAID5	Medium	High	Medium
RAID01	Low	Medium	High
RAID0	High	High	Low

TABLE I
RELATIVE QOS CLASSIFICATION OF TIERS.

B. Benchmarking with Impressions

To create realistic scenarios for our experiments, we modified Impressions [9], a benchmarking framework that generates statistically accurate filesystem images, to use our QoS attributes. We also decided to carry out our experiments on a more manageable filesystem size, since the overhead of populating a 4.5TB filesystem would be in the order of weeks.

For desired QoS, we have used only the combinations of *high*, *medium* and *low* attributes that could be matched by the storage infrastructure at hand, as shown in Table I. Using all possible combinations of attributes caused several misplaced (under or overprovisioned) data, making it hard to determine if such misplacement was caused by a defective algorithm or by the impossibility to meet a particular requirement.

We have also decided to cycle through the QoS combinations instead of selecting them at random. This proved to be simpler and more performatic than using a robust random number generator such as the Mersenne Twister [15].

C. Data Layout Analysis

The first step was to configure our test filesystem through `ifm` so that every BG was set as delivering specific QoS combinations, representing the different tiers that were identified and mapped in Table I. We could then load both the default allocator and our QoS-aware datablock allocator as discussed on Section IV and run Impressions for each one of them. Because the configuration used with Impressions was not modified during the experiments, the filesystem population happened in the exact same manner in each run.

To quantify precisely the effectiveness of the two allocators, we have used our analysis toolkit to compute the percentage of data that ends up in each category of provisioning for each QoS attribute. Table II presents these numbers with the figures on the upper part of each cell representing the default allocator and the ones on the bottom our QoS-aware datablock allocator. On this table, the "Other Space" column represents both unused space and block group metadata. We note the gains in exact QoS matching are impressive when using the QoS-aware datablock allocator (58% vs 28%).

	Very Under Prov.	**Under Prov.**	**Perf. Match**	**Over. Prov.**	**Very Over Prov.**	**Other Space**
Write Perf.	6.4 % ↓ 0 %	6.2 % ↓ 0 %	21.3 % ↓ 64.0 %	35.3 % ↓ 14.4 %	15.8 % ↓ 6.5 %	15.1 %
Read Perf.	0 % ↓ 0 %	8.1 % ↓ 0 %	40.7 % ↓ 53.7 %	20.0 % ↓ 27.6 %	16.2 % ↓ 3.6 %	15.1 %
Rel.	15.8 % ↓ 2.9 %	19.1 % ↓ 14.4 %	21.3 % ↓ 55.0 %	22.4 % ↓ 0 %	6.4 % ↓ 12.6 %	15.1 %
Avg.	7.4 % ↓ 1.0 %	11.1 % ↓ 4.8 %	27.7 % ↓ 57.6 %	25.9 % ↓ 14.0 %	12.8 % ↓ 7.5 %	15.1 %

TABLE II
QoS-MATCHING ACHIEVED USING THE DEFAULT `ext3fs` ALLOCATOR (UPPER FIGURE IN EACH CELL) AND OUR QoS-AWARE DATABLOCK ALLOCATOR (LOWER FIGURE IN EACH CELL).

VI. 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.

In this paper, we have 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 QoS.

We have prototyped this idea in a working environment by enhancing the popular Linux Extended 3 Filesystem with QoS extensions. Furthermore, we have designed and implemented working QoS-aware allocation algorithms that show convincing improvements in terms of data placement when populated with Impressions, a framework for filesystem benchmarking.

In view of the compatilibity of our work and stock `ext3fs`, one feature to be implemented is data migration. Using similar algorithms to the ones presented in this paper, we could reallocate datablocks on demand or use a background process to improve data placement during times of low utilisation. This would be relevant not only to data previously allocated with a non-QoS-aware kernel, but also to scenarios where the life-cycle of data evolves resulting in dynamic QoS requirements.

REFERENCES

[1] International Data Corporation (IDC), "The Digital Universe Decate - Are You Ready?" May 2010, http://www.emc.com/digital_universe.

[2] M. Bhadkamkar, J. Guerra, L. Useche, S. Burnett, J. Liptak, R. Rangaswami, and V. Hristidis, "BORG: Block-reORGanization for Self-optimizing Storage Systems," in *FAST '09: Proc. 7th Conference on File and Storage Technologies*, Berkeley, CA, USA, 2009, pp. 183–196.

[3] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan, "The HP AutoRAID Hierarchical Storage System," *ACM Transactions on Computer Systems*, vol. 14, no. 1, pp. 108–136, 1996.

[4] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. Veitch, "Hippodrome: Running Circles Around Storage Administration," in *FAST '02: Proc. 1st USENIX Conference on File and Storage Technologies*, Berkeley, CA, USA, 2002, p. 13.

[5] E. Anderson, S. Spence, R. Swaminathan, M. Kallahalla, and Q. Wang, "Quickly Finding Near-optimal Storage Designs," *ACM Transactions on Computer Systems*, vol. 23, no. 4, pp. 337–374, 2005.

[6] S. Akyürek and K. Salem, "Adaptive Block Rearrangement," *ACM Transactions on Computer Systems*, vol. 13, no. 2, pp. 89–121, 1995.

[7] W. W. Hsu, A. J. Smith, and H. C. Young, "The Automatic Improvement of Locality in Storage Systems," *ACM Transactions on Computer Systems*, vol. 23, no. 4, pp. 424–473, 2005.

[8] C. R. Lumb, A. Merchant, and G. A. Alvarez, "Façade: Virtual Storage Devices with Performance Guarantees," in *FAST '03: Proc. 2nd USENIX Conference on File and Storage Technologies*, Berkeley, CA, USA, 2003, pp. 131–144.

[9] N. Agrawal, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Generating Realistic Impressions for File-System Benchmarking," in *FAST '09: Proc. 7th Conference on File and Storage Technologies*, Berkeley, CA, USA, 2009, pp. 125–138.

[10] F. Franciosi and W. J. Knottenbelt, "Towards a QoS-aware Virtualised File System," in *UKPEW '09: Proc. of the 25th UK Performance Engineering Workshop*, Leeds, UK, July 2009, pp. 56–67.

[11] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel*, 3rd ed. Sebastopol, CA, USA: O'Reilly, November 2005, ch. 18.

[12] I. Dowse and D. Malone, "Recent Filesystem Optimisations in FreeBSD," in *Proc. of the USENIX Annual Technical Conference (FREENIX Track)*, Monterey, California, USA, June 2002.

[13] T. Ts'o, "e2fsprogs: Ext2/3/4 Filesystem Utilities," May 2010, http://e2fsprogs.sourceforge.net/.

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

[15] M. Matsumoto and T. Nishimura, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Transactions on Modeling and Computer Simulation*, vol. 8, pp. 3–30, January 1998.