Data Placement and Migration Strategies for Virtualised Data Storage Systems

Supervisor: Dr. William Knottenbelt

Henry Alexander Bond

June 23, 2009

Abstract

Year upon year the capacity and throughput demands for data storage are increasing. These increases have resulted in higher management overheads. The rise of cloud computing and the push back towards mainframe style centralization brings with it the need for more efficient and intelligent data storage techniques.

A significant problem with modern day file systems is that they do not take into account the characteristics of the storage devices they manage. Some storage devices perform more efficiently than others, some more reliably and others are simply more space efficient. By considering the characteristics of the storage devices and the access patterns of the data operating on them, an informed decision can be made about where space for data should be allocated. These decisions benefit the system threefold; quicker performance, greater reliability and cost saving through space efficiency.

This report details the implementation of a simulation intended on exploring the viability of such a system where data is placed intelligently depending on its source and the characteristics of its underlying storage. The system attempts to provide a framework for profiling a storage device, specifying and simulating data streams acting on the system and visualising the results.

The report demonstrates that by using by using intelligent data placement algorithms and a performance and reliability profile of the storage device, these problems can be overcome.

Acknowledgements

I would first like to give special thanks to my tutor and supervisor, Dr William J. Knottenbelt, for his unwavering support and guidance throughout the duration of my degree and this project. Secondly I would like to thank Felipe Franciosi, an Imperial College PhD student, for giving me a huge amount of help and insight into his own work. Thirdly I would like to thank my second marker, Dr Peter Harrison, for providing feedback on an early version of this report.

I would also like to thank my parents who have supported and encouraged me at every turn and finally Alfredo for his hours of invaluable advice.

Contents

Chapter 1

Introduction

1.1 Motivation

The majority of large organisations today use a relatively decentralised approach to store data where each machine will read and write data to a local storage device. Although this approach offers excellent performance it has its pitfalls. In terms of resource utilisation it turns out this approach is very inefficient. It also makes maintenance far more cumbersome since the physical locations of these devices can be widely spread.

For these reasons in recent years there has been a push back towards more centralised ‘main frame’ type storage systems. These are usually implemented in the form of Virtualised Storage Systems (VSS). A VSS is a contiguous logical volume where the notion of underlying storage devices is abstracted away. One such example of this is RAID where multiple disks are merged to provide a single abstract device for storage.

Different devices provide very different storage characteristics in terms of service time for an IO request. Ideally we would like frequently accessed data to be physically located across the high performance devices to maximise efficiency. The problem however with VSS is that the individual devices have been abstracted away, thereby making the job of efficiently placing data far more complex.

Current file systems primary aim is to reduce fragmentation and preserve directory structures. This approach optimises sequential accesses to entire files but fails however to take into account that the majority of applications access multiple files in a non sequential order.

Performance is important but it is not the only characteristic to take into account when using VSS's. Reliability of data is another very important aspect. If a disk fails is the data still recoverable? If hardware is stolen will confidential data be compromised?

Space efficiency is another important aspect of VSS's. Storage is finite and the volume data produced is growing year upon year - 'Total disk storage systems capacity shipped reach 1,777 petabytes, growing 43.7% year over year'[1]. It is therefore important to keep data compact wherever possible.

Throughout this report I will refer to these data considerations as Quality of Service (QoS). For example a bank's financial data will have to be stored in a fashion that guarantees high reliability. A temporary file however probably needs only high performance. These two pieces of data have very different QoS attributes.

Allocation of data is a very important aspect of this problem but the quality of service attributes are not static and will change over time for different data streams. This implies that not only allocation must be considered but also migration, to keep the underlying data in the most suitable areas possible.

1.2 Approach

There are a few ways to investigate this problem. One extreme would be to fully implement a file system as a Linux module which could be loaded into the Linux kernel. This file system would take into account QoS of data and the performance characteristics of the underlying device. The other extreme would be to do a simulation of such a file system. Although the file system approach would be the most useful solution, due to the difficultly of the problem and the time constraints, I decided to choose the simulation.

This report details a simulation of a system which allocates and mi-

[1] This statistic is from: `http://www.idc.com/getdoc.jsp?containerId=prUS21411908`

grates data depending on the characteristics of a specified underlying VSS. Specifically I present a framework which:

- The simulation profiles the VSS's logical address space to create a QoS model of the device. See chapter 3.
- Allows the specification of data streams and their characteristics. See chapter 4.
- Simulates the streams operating on the model using allocation and migration techniques. See chapter 4.
- Visualises results by shoing performance graphs and location of data. See chapter 5.

Chapter 2

Background

2.1 Storage Devices and Techniques

Today the most common type of storage medium, from large data centres to the home computer, is the conventional hard disk drive. This device consists of a mechanical read/write head which moves over a spinning disk made of ferromagnetic material. When an I/O request is issued the read/write head will magnetize/detect magnetism depending on the request. It uses this technique to read/write a series of bits representing information. A disk will typically consist of sectors, tracks and zones as demonstrated in the figure 2.1[1].

[1] This diagram came from: `http://searchoracle.techtarget.com/digitalguide/images/Misc/disk_layout.jpg`

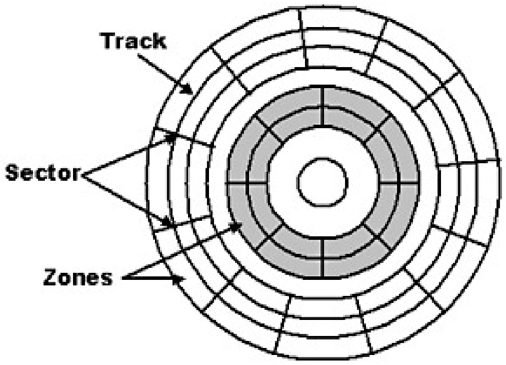


Figure 2.1: Simplified layout of a magnetic hard disk

One of the most significant bottlenecks in computing today is I/O. Computers can process data faster than they can read or store it. This limits a computer's potential and can be a major deficit on performance.

There are various ways to reduce the problem imposed by this bottleneck. A typical hard drive will spin at a rate of 7,200rpm however some more sophisticated drives will spin at 10,000 rpm. The faster the spin the faster data can be read from and written to the drive and the less significant this bottleneck becomes. Another method to improve performance is RAID (Redundant Array of Inexpensive Disks). There are two main ideas to RAID:

1. Improve I/O performance *by striping data*
2. Improve reliability of stored data *by adding redundancy*

RAID can be done on many different levels. A brief description of some of the levels has been included below[2].

RAID Level 0: 'Distributes data across several disks in a way that gives improved speed and full capacity, but all data on all disks will be lost if any one disk fails.'

[2] The raid descriptions and images have have been taken from: `http://en.wikipedia.org/wiki/Redundant_array_of_independent_disks`

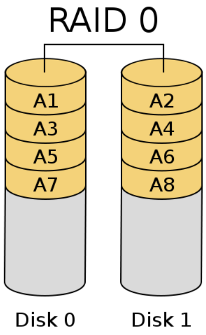


Figure 2.2: RAID 0

RAID 1: 'Could be described as a real-time backup solution. Two (or more) disks each store exactly the same data, at the same time, and at all times. Data is not lost as long as one disk survives. Total capacity of the array is simply the capacity of one disk. At any given instant, each disk in the array is simply identical to every other disk in the array'.

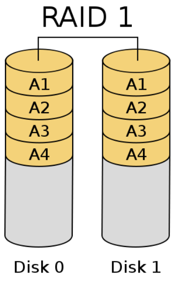


Figure 2.3: RAID 1

RAID 5: 'Combines three or more disks in a way that protects data against loss of any one disk; the storage capacity of the array is reduced by one disk'.

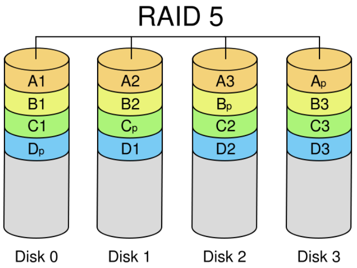


Figure 2.4: RAID 5

RAID 6: ‘Can recover from the loss of two disks - *As RAID 5 with 2 parity stripes*’

RAID 10: ‘Uses both striping and mirroring. “01” or “0+1” is sometimes distinguished from “10” or “1+0”: a striped set of mirrored subsets and a mirrored set of striped subsets are both valid, but distinct, configurations’.

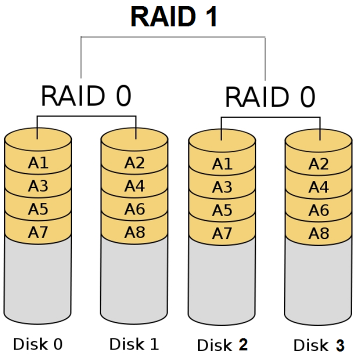


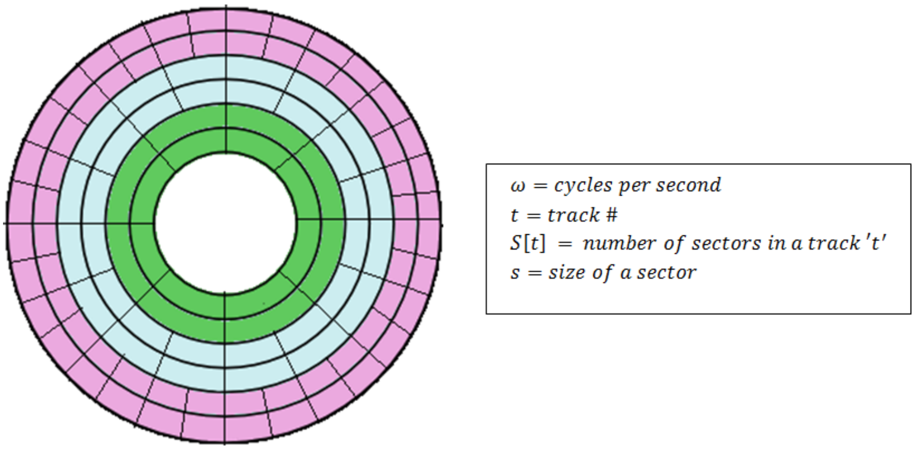
Figure 2.5: RAID 10

RAID can be implemented using either hardware or software. Hardware RAID has the advantage of not using the local processor or memory. It is more expensive but offers better performance.

Although RAID improves performance it does not take into account any performance profiles of its underlying data storage devices. One might expect the service time on a conventional hard drive for each sector to be constant. This however is not the case. If a read at LBA 0 takes 1 time unit this does not imply that a read at LBA 10,000 will also take 1 time unit.

2.2 Storage Device Characteristics

Figure 2.6 is an over simplified representation of the layout of a hard disk. The zones are differentiated by different colours. You will notice zones further from the centre have a greater number of sectors and therefore a greater storage capacity.



$\omega = cycles\ per\ second$

$t = track\ \#$

$S[t]\ =\ number\ of\ sectors\ in\ a\ track\ 't'$

$s = size\ of\ a\ sector$

Figure 2.6: Simplified view of a hard disk with algebraic definitions

$$S[t+1] \geq S[t]$$

A result of this is, I/O operations on the outer tracks are more efficient than the inner tracks, since the rotational speed remains constant and the linear speed varies.

$$\omega \times S[t] \times s = I/O\ per\ second$$

So for increasing t I/O efficiency increases. This is the reason for the nonlinear relationship between the LBA and the throughput.

Figure 2.7 is a plot of the throughput[3] of a standard magnetic hard drive across its address space.

[3] data size of request/time to service request

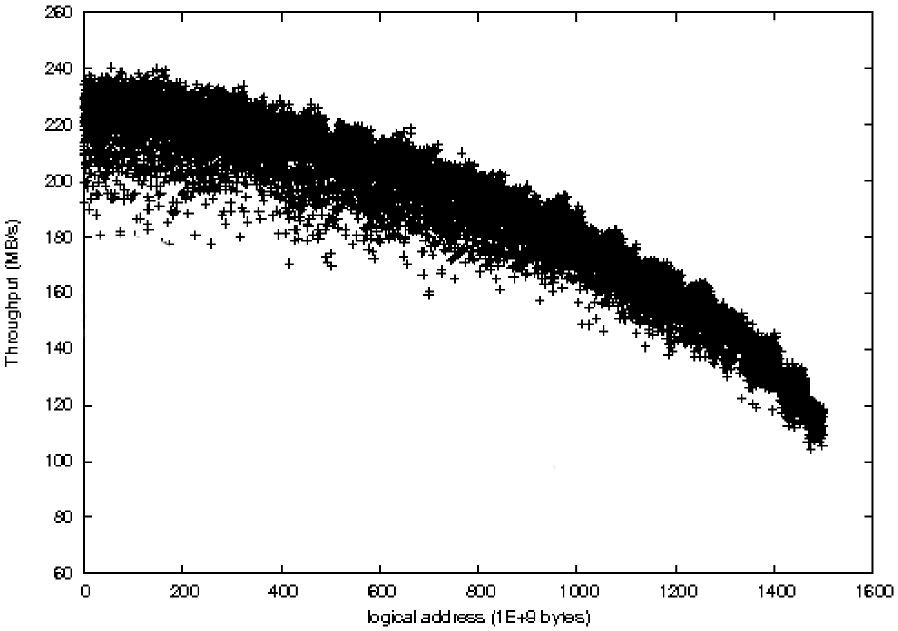


Figure 2.7: Standard Magnetic Hard Drive Performance Profile

Of course hard disks can be set up in many different configurations for example RAID (as mentioned earlier). This will change this curve however the general idea that the outside regions of the disks have a greater number of sectors will still dictate where the high throughput will occur.

In recent years flash memory is starting to become far more prominent. 'Flash memory stores information in an array of memory cells made from floating-gate transistors'[4]. Flash memory has the advantage of not having any mechanical/moving parts. This makes its operation far faster and the device as a whole less volatile. Since it has no moving parts its performance profile is flat.

2.3 Virtualised Storage Systems and LVM

When it comes to virtualised storage systems this relationship between logical location and throughput becomes more complex. The reason being that depending on the underlying devices, you get far more variance between the

[4] This description came from: `http://en.wikipedia.org/wiki/Flash_memory`

logical location and the throughput. Figure 2.8 shows 2 VSS's spread over an infrastructure consisting of 4 different tiers.

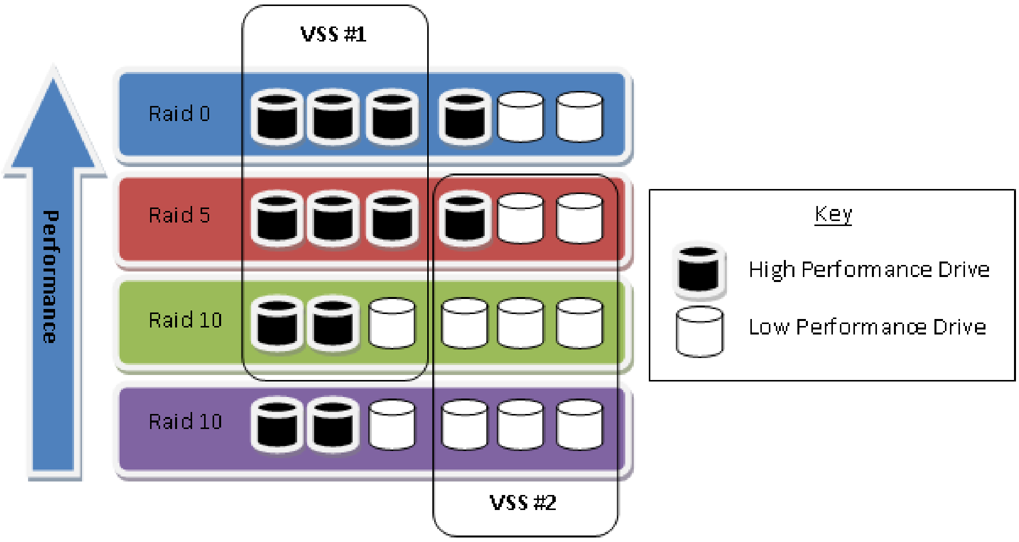


Figure 2.8: Shows 2 VSS's spread over 4 different storage tiers ordered by performance

VSS #1 and #0 spread across different areas of the available tiers and inherently provide different QoS attributes. VSS #1 will have a very good performance profile due to the majority of its underlying storage devices being of high performance and in fast raid configurations. VSS #0 however, does not provide such good performance but it does provide good data reliability, as every single tier it spreads over keeps redundancy data. This shows how QoS can change depending on the underlying media.

In general an end user should not need to be concerned with the aspects of the separate storage devices and where (physically) the data resides. Linux provides a useful kernel utility called LVM (Logical Volume Manager) which abstracts the notion of individual logical drives away from the user by combining them into a virtually contiguous logical device. In other words it is an implementation of a VSS scheme. Figure 2.9 shows a performance profile over a sample LVM.

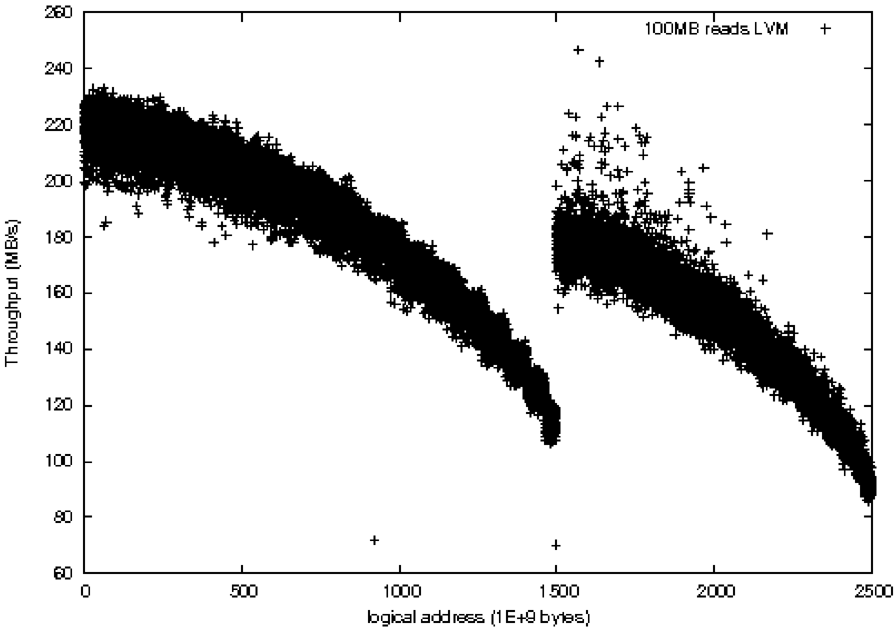


Figure 2.9: Profile of an LVM consisting of a RAID 5 configuration and a RAID 10 configuration

Suppose we have two sets of data ‘A’ and ‘B’. It is known that data ‘A’ is accessed 20 times more than data ‘B’ per unit time. For the best performance we would desire A to be in an area of high throughput and B to be in an area of low throughput. In terms of figure 2.9 we would prefer data ‘A’ to lie between 0-500 and data ‘B’ to lie between 1000-1500. If there was no room for data ‘A’ in a high performance area we would prefer it to be placed in a low performance area (or the best available) and to be migrated to a high performance area when space became available.

2.4 Data Streams Characteristics

If you were to investigate how individual processes perform I/O over a period of time you will find that they are very deterministic in the way they do so. A research project BORG [1] has built an extension to the Linux VFS to exploit this fact to improve I/O performance.

Application	CPU (s)	I/O waits (s)	Seq I/O (%)
firefox	1.56	3.71	51.08%
oowriter	3.35	7.93	60.99%
xemacs	0.92	5.94	65.35%
xinit	0.57	3.55	67.42%
acroread	0.99	5.08	56.55%
eclipse	15.42	14.88	55.07%

Figure 2.10: Table of CPU and memory access data for a number of applications from the BORG paper [1])

A main difference between the BORG project and that which I am investigating is that I will be talking about data streams (as an abstract concept) where BORG talks about processes specifically. If we know the characterstics of a data stream we can exploit the system to give us better performance. Consider the following situation:

Stream A and Stream B both want to allocate space on the same VSS:

Data Stream A	**Data Stream B**
Stream A is system critical Over half of its I/O operation are reads. All I/O operations are sequential I/O operations are typically small	Stream B is non critical All of its I/O is writes. All I/O operations are non-sequential I/O operations are typically large

Figure 2.11: QoS situation for 2 data streams

The most suitable place on the logical partition for data from stream A is very likely going to be different from the most suitable place for stream B. If the operating system had a performance profile of the device it could make better decisions on where to place this data and thereby increase efficiency.

The 4 main aspects to consider when dealing with data placement are the following:

1. Required I/O performance
   - The speed at which the stream requires data to be read or written.
2. Required Data Reliability

- I am going to define reliability to mean the following - *probability data will not be permanently lost in the event of a hardware failure.*

3. Required Space Efficiency

   - No data store is unlimited in size and so it is important to store data as compactly as possible without impacting performance.

4. Expected Data Growth

   - Some data is more likely to grow than others. It is important to take this into consideration since data fragmentation happens when there is not enough spare contiguous space after the initial allocation. This can degrade performance.

Each RAID level offers different trade-offs between the aspects I am considering. For example RAID 0 offers very high performance as disk accesses can be performed in parallel. Space efficiency is also very good since it stores no redundant data. However in terms of reliability it is very poor. If one of the drives fails all data is effectively lost.

On the other hand RAID 1 offers very good reliability as if one of the drives is lost all data can be recovered. It also offers very good read performance as either disk can be accessed for a particular piece of data (depending on which one is available). Write performance is degraded however, as the 1 request must be serviced by both disks

The chart below demonstrates these trade-offs for some of the different RAID levels:

RAID	Read I/O	Write I/O	Data Reliability	Space Efficiency
0	H	H	L	H
1	H	L	M	L
5	M	L/M	H	M
10	M	L/M	H	M

Figure 2.12: RAID Performance Chart

A VSS will typically span across multiple types of disks and RAID levels so it is important to know the charactersic of the storage to get the best mapping for the Qos attributes.

The useful thing about keeping QoS attributes for data is that they can be changed depending on the situation. For example an organisation may have a very heavy workload one week so performance would be paramount. The attributes could easily be changed and the data would be migrated to give better performance. The next week perhaps there is a scheduled inspection. The QoS attributes of data could now be changed to high reliability and the data would be migrated accordingly to reduce the risk of data being lost/compromised.

2.5 Similar Work

The following projects investigate similar ideas in terms of intelligent data placement:

BORG [1] (Block-reORGanization and Self-optimization in Storage Systems) is a module for the linux kernel which sits between the file system layer (etc3, jfs, ntfs etc...) and the I/O scheduler. It constantly evaluates process access patterns based on temporal, process-level, and block-level attributes. It then constructs access pattern graphs which allows for more intelligent data placement.

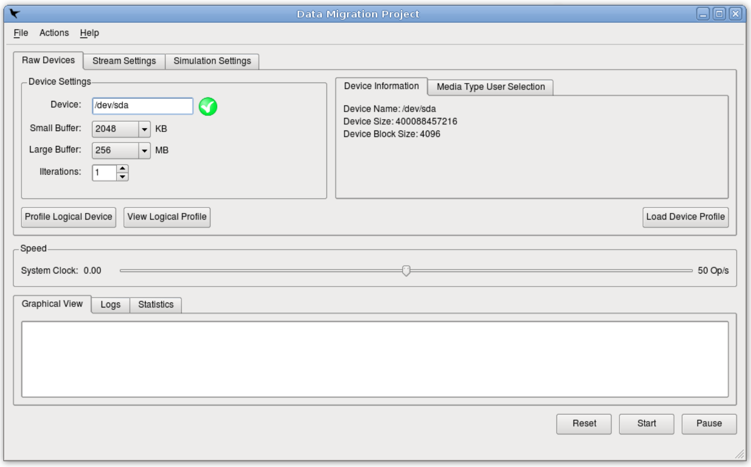
Data Placement and Migration in Virtualised Storage Systems [2] (Felipe Franciosi Imperial College). The task of this project is to modify the well known file system ext3 to include QoS attributes. Ext3 leaves some redundant bits for user modification in the individual inodes. The intent of this project is to use these bits to mark QoS attributes for sets of data. The system will use these attributes along with a performance profile of the underlying storage devices to intelligently place data.

Chapter 3

Profiling and Processing

3.1 Profiling

In order to simulate a storage device, a performance, reliability and space efficiency profile of said device is required. Storage devices will perform differently under different circumstances, for example large writes will usually take longer than small reads. Not only do devices vary in how they perform but also in terms of reliability (chance of failure) and space efficiency. For these reasons it is necessary to take various profiles into account to get a complete picture of how the device performs.



Device: /dev/sda

Small Buffer: 2048 KB

Large Buffer: 256 MB

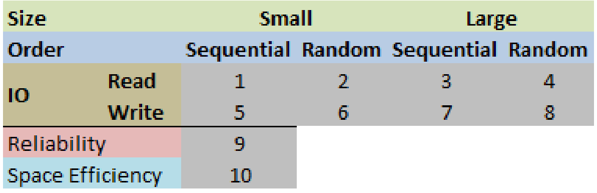
Iterations: 1

Device Name: /dev/sda
Device Size: 400088457216
Device Block Size: 4096

System Clock: 0.00 50 Op/s

Figure 3.1: The GUI of the profiling utility

My implementation provides a utility for creating, visualising and processing a set of 10 profiles based on 5 characteristics [Order of IO, Size of IO, Type of IO, Reliability of Device, Space Efficiency of Device]. Figure 3.1 shows the GUI for profiling the device.



Size		Small		Large	
Order		Sequential	Random	Sequential	Random
IO	Read	1	2	3	4
	Write	5	6	7	8
Reliability		9			
Space Efficiency		10			

Figure 3.2: Table showing the 10 different type of profiles that are created by the utility

The profiling application will create 10 different profiles. Figure 3.2 shows how each profile is composed. Profiles 1 – 8 all describe IO performance of the device and are composed of [Order of IO, Size of IO, Type of IO]. Figure 3.3 and 3.4 describe the method used to achieve sequential and random IO.

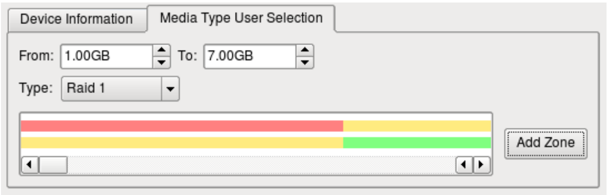
```
sequential_profile(Device dev, int buffer_size) {
        list<int> result
        Timer t
        dev.seek(START)
        do {
                t.start()
                dev.io()
                t.stop()
                result.add(t.time)
        } while( dev.seek(buffer_size) )
        output result
}
```

Figure 3.3: Pseudo-code for how the utility conducts a sequential profile

```
random_profile(Device dev, int buffer_size) {
        list<int> result
        RandomBag rb( dev.size / buffer_size )
        Timer t
        while(rb.size > 0) {
                int lba = rb.pick() * buffer_size
                t.start()
                dev.seek(lba)
                dev.io()
                t.stop()
                result.add(t.time)
        }
        output result
}
```

Figure 3.4: Pseudo-code for how the utility conducts a random profile

One problem with storage devices is that they can't report what type of device they are. For example device '/dev/md1' may be a raid 5 device however this is abstracted from the system. When it comes to reliability and space efficiency this is a problem since these attributes completely depend on the configuration of the device. A result of this is that the profiler cannot automate profiles 9 and 10 from figure 3.2. My implementation provides a utility for manually choosing the type of device by selecting a range in GB. Figure 3.5 shows an example of a VSS where ranges 0GB – 7GB describes a Raid 1 device and 7GB onwards describes a Raid 5 device.



From: 1.00GB To: 7.00GB

Type: Raid 1

Figure 3.5: The GUI for selecting device types within logical ranges ranges

The colored strips in figure 3.5 represent space efficiency (upper) and reliability (lower). Where red represents low, yellow medium and green high.

3.2 Processing

Profiling the device yields a set of raw data which needs to be transformed and processed in order to make it suitable for simulation. This processing comes in a series of steps illustrated in figure 3.6.

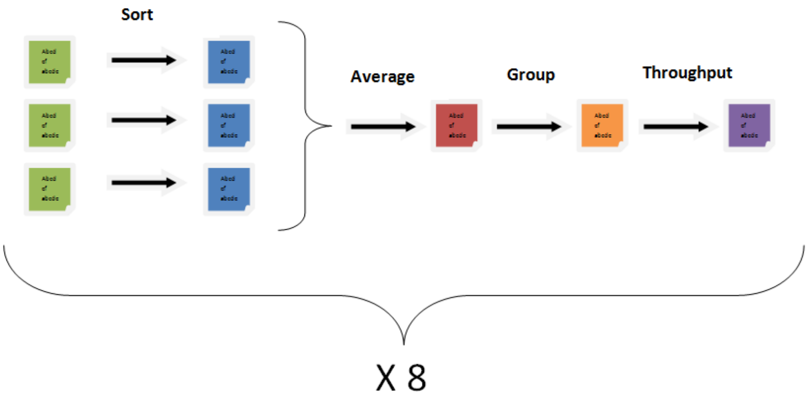


Figure 3.6: Pictoral representation of the operations performed while processing the profiling data

1. Sorting – Random profiles will not be in block address order. They must first be sorted (this is done using linux's sort utility).
2. Averaging – The profiler is set to do a specified number of iterations of profiles to get the most accurate model, these must be averaged into a single file.
3. Grouping – The filesystem (detailed in section 4.2) requires a certain set block group size. The profiles must be quantized into this size by combining (Summing) the results from multiple adjacent elements.
4. Throughput – Throughput (kb/s) is a more useful representation than time.

Also created are two extra files classifying Reliability and Space Efficiency as High Medium and Low for each block group[1].

[1] Block Group – defined in section 4.2

Chapter 4

Simulation

This chapter describes the main part of the application. This section talks about the notion of data streams and how they operate on the underlying storage intelligently. Also described is a simple file system and various placement and migration techniques. Appendix A demonstrates a UML diagram of the simulation.

4.1 Data Streams

In real systems a ‘sea’ of requests will be made to the storage from a variety of sources. For example the system may be a file server containing users home directories, it may also be a database server storing experimental data. Both examples will almost certainly have different access patterns and so requests from these sources should be treated differently. In order to simulate such situations I have abstracted these notions of access sources into data streams.

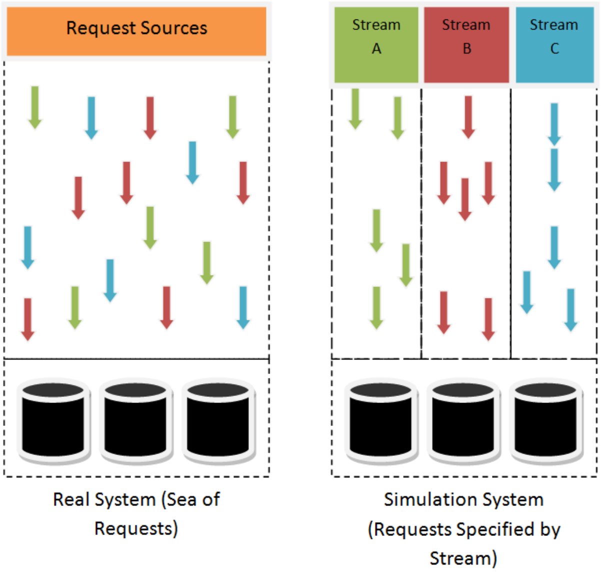
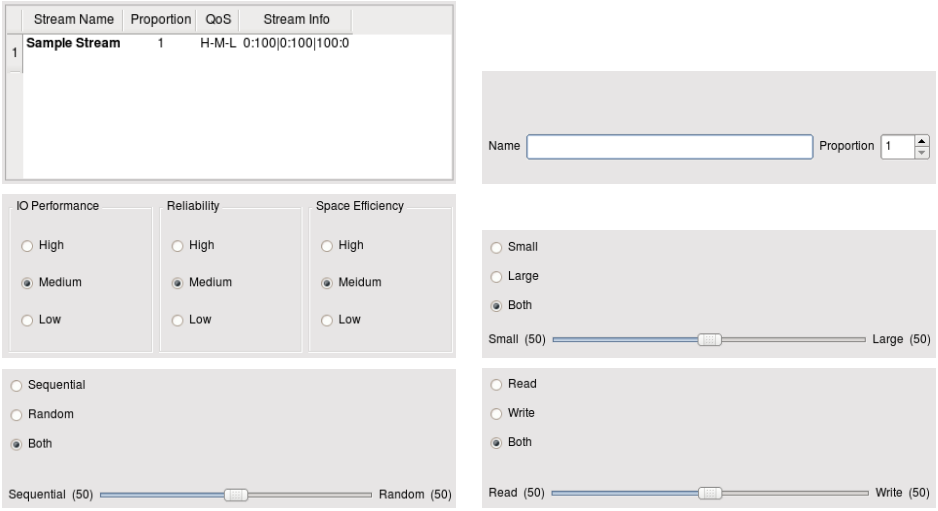

Request Sources
Stream A
Stream B
Stream C
Real System (Sea of Requests)
Simulation System (Requests Specified by Stream)

Figure 4.1: A comparison between a real system where requests are ungrouped and a simulation where requests are grouped

My implementation provides a utility for specifying data streams according to their individual access patterns and required quality of service. Figure 4.2 demonstrates how settings for a stream can be selected.



	Stream Name	Proportion	QoS	Stream Info
1	**Sample Stream**	1	H-M-L	0:100|0:100|100:0

Name □ Proportion 1

IO Performance
○ High
● Medium
○ Low

Reliability
○ High
● Medium
○ Low

Space Efficiency
○ High
● Meidum
○ Low

○ Sequential
○ Random
● Both

Sequential (50) Random (50)

○ Small
○ Large
● Both

Small (50) Large (50)

○ Read
○ Write
● Both

Read (50) Write (50)

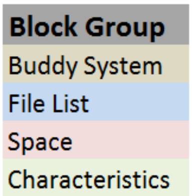
Figure 4.2: The GUI for the Stream Selection utility

- Stream Characteristics
  - Name: A name is specified to identify the stream (i.e. Database)
  - Proportion: Sets the proportion of requests made by the stream relative to the other streams specified
- Quality of Service
  - IO Performance: Specifies the level of IO performance required (High Medium or Low)
  - Reliability: Specifies the level of Reliability required (High Medium or Low)
  - Space Efficiency: Specifies the level of Space Efficiency required (High Medium or Low)
- Access Patterns
  - Data Size: Specifies the expected proportion of Small:Large data accesses made by the stream
  - Order of Access: Specifies the expected proportion of Sequential:Random data accesses made by the stream
  - IO Type: Specifies the expected proportion of Read:Write data accesses made by the stream

4.2 File System

Since the point of this simulation is to manage data allocation, a file system is necessary. The initial idea was to have an in memory data structure to hold my file system. This became a problem however, since this simulation should be able to represent very large storage systems (TB's in size) the structure would not be able to fit into main memory. To solve this problem I designed a cache which deals with block groups.

Block groups hold meta data about fixed portions (256MB) of the file system. Figure 4.3 shows a high level representation of what fields the block groups contain.



Block Group

Buddy System

File List

Space

Characteristics

Figure 4.3: Pictoral representation of a Block Group

The first field to notice is the buddy system. To reduce external fragmentation I decided to use the buddy memory allocation system. This works in the following way. Storage is represented by an asymmetric 2 dimensional matrix where each level of the matrix holds an array of bits corresponding to a particular data size. A unset bit implies storage is available in that section and a set bit implies that the storage is occupied. Allocation works as follows[1].

1. If memory is to be allocated
   (a) Look for a memory slot of a suitable size (the minimal 2k block that is larger or equal to that of the requested memory)
   (b) If it is found, it is allocated to the program
   (c) If not, it tries to make a suitable memory slot. The system does so by trying the following:

[1] Method description taken from `http://en.wikipedia.org/wiki/Buddy_memory_allocation`

i. Split a free memory slot larger than the requested memory size into half

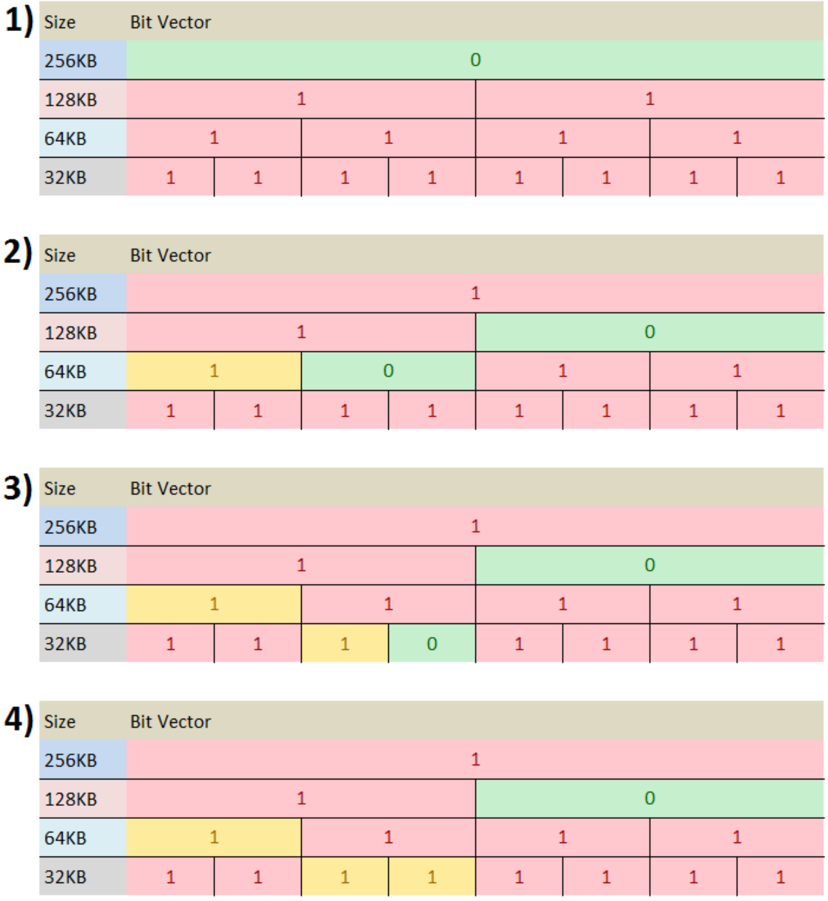
ii. If the lower limit is reached, then allocate that amount of memory otherwise go back to step a)

2. If memory is to be freed

   (a) Free the block of memory

   (b) Look at the neighbouring block - is it free too?

   (c) If it is, combine the two, and go back to step 2 and repeat this process until either the upper limit is reached (all memory is freed), or until a non-free neighbor block is encountered



1)

<table>
<tr><th>Size</th><th colspan="8">Bit Vector</th></tr>
<tr><td>256KB</td><td colspan="8">0</td></tr>
<tr><td>128KB</td><td colspan="4">1</td><td colspan="4">1</td></tr>
<tr><td>64KB</td><td colspan="2">1</td><td colspan="2">1</td><td colspan="2">1</td><td colspan="2">1</td></tr>
<tr><td>32KB</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td></tr>
</table>

2)

<table>
<tr><th>Size</th><th colspan="8">Bit Vector</th></tr>
<tr><td>256KB</td><td colspan="8">1</td></tr>
<tr><td>128KB</td><td colspan="4">1</td><td colspan="4">0</td></tr>
<tr><td>64KB</td><td colspan="2">1</td><td colspan="2">0</td><td colspan="2">1</td><td colspan="2">1</td></tr>
<tr><td>32KB</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td></tr>
</table>

3)

<table>
<tr><th>Size</th><th colspan="8">Bit Vector</th></tr>
<tr><td>256KB</td><td colspan="8">1</td></tr>
<tr><td>128KB</td><td colspan="4">1</td><td colspan="4">0</td></tr>
<tr><td>64KB</td><td colspan="2">1</td><td colspan="2">1</td><td colspan="2">1</td><td colspan="2">1</td></tr>
<tr><td>32KB</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td></tr>
</table>

4)

<table>
<tr><th>Size</th><th colspan="8">Bit Vector</th></tr>
<tr><td>256KB</td><td colspan="8">1</td></tr>
<tr><td>128KB</td><td colspan="4">1</td><td colspan="4">0</td></tr>
<tr><td>64KB</td><td colspan="2">1</td><td colspan="2">1</td><td colspan="2">1</td><td colspan="2">1</td></tr>
<tr><td>32KB</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td></tr>
</table>

Figure 4.4: Diagram of the Buddy Allocation System of a particular block group undergoing various operations

Figure 4.4 shows a case study where a hypothetical block group is undergoing various operations. In stage 1) all memory is currently available denoted by the fact the bit in the top level is unset. At stage 2) a request is made for a 64kb block of data. Since there are no free 64kb blocks this is achieved by 'splitting' the 256kb block into 2 128kb blocks, and further splitting 1 of the 128kb blocks into 2 64kb blocks. At stage 3) a request is made for a 32kb block. Since once again there are no free 32kb blocks this is achieved by splitting the remaining 64kb block into 2 32kb blocks. Finally in stage 4) a request is made for another 32kb block. Since there is already one available this is used. This diagram can also be read in reverse where the original state of the buddy system is 1 free 32kb block. As space is reclaimed adjacent unset bits coalesce in the higher levels eventually leading back to stage 1).

The 'File List' field is a list of the files contained on the block group including the size of the file and the start address on the storage device. This is necessary in order to find a file for modification, deletion or migration.

The space field is an integer denoting the amount of free storage left on the block group. This is used during allocation to check if there is enough room for a particular piece of data to lie.

The final field, characteristics, is a composite field containing profiling information collected from the profiling utility described in section 3.1. The fields are composed of combinations of Large/Small, Sequential/Random, Read/Write. An advantage of representing these attributes on a block group by block group basis is that it quantises the profile of the logical device removing outliers.

As previously mentioned in many cases there will be too many pages to store in main memory so a cache is necessary. I decided to implement a simple round robin cache. In order to retrieve a block group, the cache calculates *pagenumber 'mod' cahcesize* and checks if the block group is in memory at that location. If not it writes the current block group back to storage and retrieves the desired one.

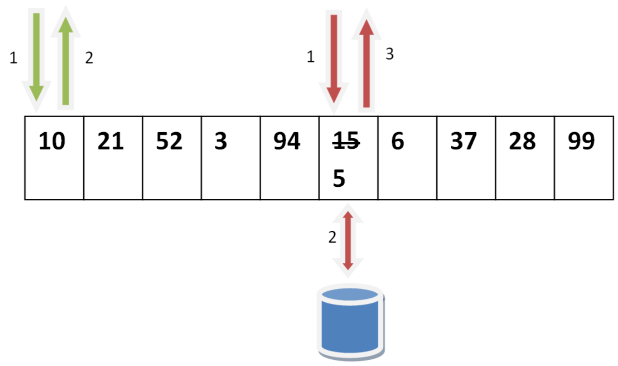


Figure 4.5: Pictoral representation of the cache undergoing various operations

Figure 4.5 shows an example of cache set with a size of 10. The cache has already been running for a while since it is full of block groups. The diagram shows two situtations:

- Green: The page 10 is requested, since $10\ `mod'\ 10 = 0$ the page 10 should be found in location 0. The page is then returned.
- Red: The page 5 is requested however the page 5 is not found in location 5. The cache instead writes 15 back to disk and retrieves and returns 5.

Figure 4.6 shows the code used for retrieving an writing pages to and from the cache.

```
input_file = new ifstream(/*file*/, ios::binary);
output_file = new ofstream(/*file*/, ios::binary);
BuddyPage* BuddyCache::requestPage(int page)
{
        int offset = page % BC_BUFFERSIZE;
        if(which_page[offset] != page)
        {
                if(which_page[offset] > -1)
                        write_to_memory(
                                which_page[offset],
                                buddy_cache[offset]);
                seek((fstream*)input_file,page);
                buddy_cache[offset] =
                        new BuddyPage(*input_file);
                which_page[offset] = page;
                return buddy_cache[offset];
        }else return buddy_cache[offset];
}
void BuddyCache::write_to_memory(int p, BuddyPage* bp)
{
        seek((fstream*)output_file,page);
        buddy_page->serialize(*output_file);
        delete buddy_page;
}
void BuddyCache::seek(fstream* stream, int page)
{
        stream->seekg(page * sizeof(BuddyPage));
}
```

Figure 4.6: C++ code for serializing, reading and writing ‘Block Groups’ in the cache

4.3 Events and Event Generation

In a real system requests would be sent to the storage device. In order to simulate this I designed an events generator to sent requests to the storage. Each stream which has been specified, as described in section 4.1, will affect the type and frequency of events generated. In order to achieve realistic results there are three main points to consider:

1. Which stream triggers the event
2. What type of event is triggered
3. Amount of time until the next event

The simplest problem is picking the appropriate stream in each iteration of the simulation. The likelihood of a stream being chosen depends directly on its proportion. As a result of this the probability of a particular stream being selected is $\frac{proportion(stream)}{\sum proportion(i)}$.

Suppose a situation where there are three streams A, B and C with proportions 1, 2 and 3 respectively. The probabilities of each stream being selected is as follows:

Stream	Fraction	Probability
A	$\frac{1}{1+2+3}$	0.17
B	$\frac{2}{1+2+3}$	0.33
C	$\frac{3}{1+2+3}$	0.5

Figure 4.7: Probabilities of a stream being selected

In order to choose a stream from this distribution I use the method described by the pseudo-code described in figure 4.8. A random number is selected between [0,1), each stream is then selected in turn and the probability of each stream is subtracted from the random number. At the point where the number drops to 0 or below the loop exits and the current stream is selected.

```
choose_stream(Stream[] streams,
                   float[] probabilities) {
         float random = rand()
         for( i = 0, random > 0, i++)
                random -= probabilities[i]
         output streams[i]
}
```

Figure 4.8: Pseudo-code for selecting a stream based on a set of probabilities

After the stream has been selected the type of event issued by that stream needs to be selected. I distinguish between three types of event:

1. Creation – The introduction of a section of data into the file system
2. Deletion – The removal of a section of data from the file system
3. Modification – The modification to a section of data in the file system

Since this is a simulation of the system and there are no real data streams it is important to produce requests as closely to a real system as possible. The interesting cases of the simulation occur when the file system is at various capacities. It is therefore important to be able to control the capacity at which the file system should grow to during the simulation. In order to achieve this capacity the events triggered should depend on the current free space and desired free space.

A creation event will cause the system to become more full, whereas a deletion event will cause the system to become more empty. A modification event however won't affect the free space at all. The current capacity should therefore change the probabilities of each event in the following ways:

$$\begin{aligned}
\text{System is fuller than desired} &\longrightarrow P(delete) > P(creation) \\
\text{System is at desired state} &\longrightarrow P(delete) = P(creation) \\
\text{System is emptier than desired} &\longrightarrow P(delete) < P(creation)
\end{aligned}$$

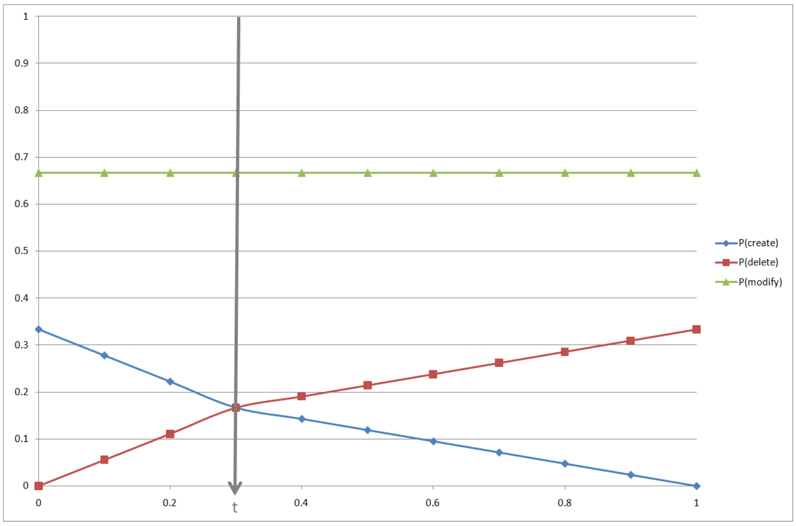


Figure 4.9: Chart plotting system capacity against the probability of each event type occurring, with a target capacity t

$$P(modify) = k_{modify} \tag{4.1}$$

$$P(create) = 1 - P(delete) \tag{4.2}$$

$$P(delete) = \begin{cases} \frac{x \times k_{delete}}{2t} & capacity \leq t \\ \frac{(x-t)k_{delete}}{2(1-t)} + \frac{k_{delete}}{2} & Otherwise \end{cases} \tag{4.3}$$

Figure 4.10: Equations of lines plotted in figure 4.9

If I assume linearity this defines my function quite easily. The chart in figure 4.9 shows how the probabilities of each event vary with time. Figure 4.10 shows the equations of each of the lines. The constants k_{modify}, k_{create} and k_{delete} and pre defined constants set to $\frac{2}{3}$, $\frac{1}{6}$ and $\frac{1}{6}$ respectively in the implementation. Notice how when the capacity is at target 't' the probability of a creation and a deletion is equal therefore keeping the filesystem around this set capacity.

Finally if the type of event chosen happens to be a modify event then the way in which the data is read must conform with the set characteristics of the stream (Data Size, Order and IO type).

When an event is 'fired' the expected time of the event is calculated and the simulation clock is advanced. These times are calculated by working out a weighted average of the 8 performance profiles contained in each block group and multiplying this by the size of data accessed. These times are written to a log file which can be run through the visualisation engine described in chapter 5.

In queuing theory the Poisson distribution represents the probability that a certain number of events will occur in a fixed time period. The exponential distribution represents the probabilities of the time between events.

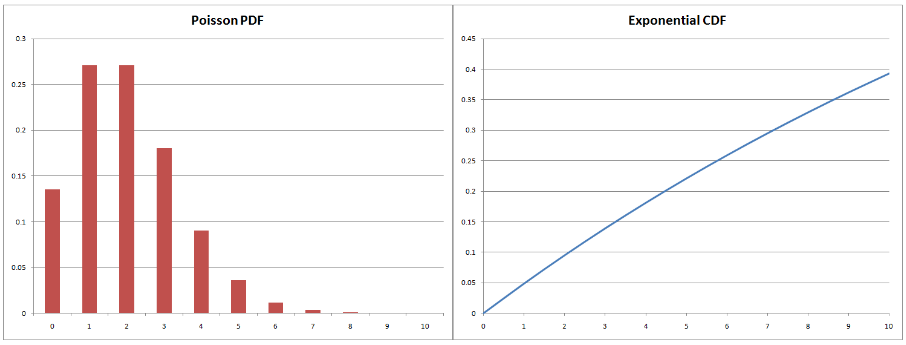


Figure 4.11: Graph showing the Poisson distributions PDF (left) and Exponential distributions CDF (right)

Since I would like to 'fire' my events realistically I will use the exponential distribution to determine the time between the current event and the next event. The following equation describes the CDF of the exponential distribution:

$$P(t, \lambda) = 1 - e^{-\lambda t}$$

To represent this particular problem I have chosen my rate variable λ to be the $\sum(proportions)$. To generate a time period however the distribution must be in terms of t.

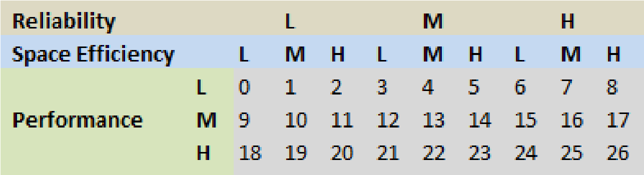
$$x = 1 - e^{-\lambda t}$$
$$t = -\frac{1}{\lambda} ln(1 - x)$$

By generating random numbers for x in the range [0,1) I can generate time periods between the next event.

4.4 Placement and Migration

4.4.1 Placement

In order to find the most appropriate location to allocate space for the data streams it is necessary to quantify how each block group performs under the three different attributes. Data Throughput, Space Efficiency and Reliability. Figure 4.12 shows how each combination is classified.



Reliability		L			M			H		
Space Efficiency		L	M	H	L	M	H	L	M	H
Performance	L	0	1	2	3	4	5	6	7	8
	M	9	10	11	12	13	14	15	16	17
	H	18	19	20	21	22	23	24	25	26

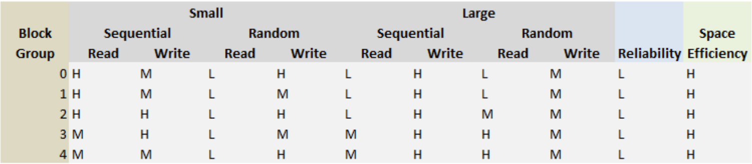
Figure 4.12: Table representing a function which takes a combinations of QoS attributes to a natural number

To classify each page as such it is necessary to find a weighted combination of the throughputs. Each stream will have a separate classification for each block group since each stream has different constants (large/small, sequential/random, reads/writes).

```
weighted_combination(double[][] inputs,
                                  float[] weights) {
        double[] result
        for( length = 0 -> inputs[].length ) {
                for( profile = 0 -> inputs.length ) {
                        result[length] +=
                                inputs[profile][length]
                                * weights[profile]
                }
        }
        output result
}
```

Figure 4.13: Pseudo-code for finding weighted combination of throughputs from the profiles

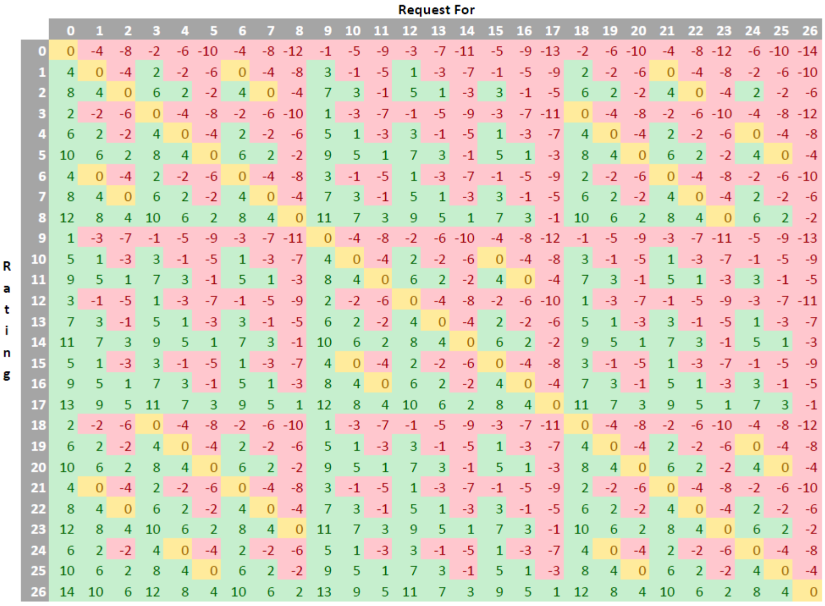
It is then necessary to classify these throughputs into Low, Medium or High. These classifications should be relative to the underlying storage and not absolute since the profiler should not be dependent on the system it runs. I classify them by finding the throughput which lies on the 33^{rd} percentile and the 66^{th} percentile. Any throughput which lies before the 33^{rd} percentile should be classified as low, between the 33^{rd} percentile and the 66^{th} percentile should be classified as Medium and anything greater than the 66^{th} percentile should be classified as High.



	Small				Large					
Block	Sequential		Random		Sequential		Random			Space
Group	Read	Write	Read	Write	Read	Write	Read	Write	Reliability	Efficiency
0	H	M	L	H	L	H	L	M	L	H
1	H	M	L	M	L	H	L	M	L	H
2	H	H	L	H	L	H	M	M	L	H
3	M	H	L	M	M	H	H	M	L	H
4	M	M	L	H	M	H	H	M	L	H

Figure 4.14: Table displaying the first 5 block groups of a sample profile quantised into L M H

Figure 4.14 shows an example profile where all the attributes have been translated into L M H's. The combinations of L M H's can now be easily transformed into the classifications 0 – 26. Of course each stream will value each classification differently depending on its performance weighting. In order to work out how each stream values a particular rating it is first necessary to calculate a 'request receive' matrix based on constants which value Throughput, Reliability and Space Efficiency against one another.



Request For

Rating	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
0	0	-4	-8	-2	-6	-10	-4	-8	-12	-1	-5	-9	-3	-7	-11	-5	-9	-13	-2	-6	-10	-4	-8	-12	-6	-10	-14
1	4	0	-4	2	-2	-6	0	-4	-8	3	-1	-5	1	-3	-7	-1	-5	-9	2	-2	-6	0	-4	-8	-2	-6	-10
2	8	4	0	6	2	-2	4	0	-4	7	3	-1	5	1	-3	3	-1	-5	6	2	-2	4	0	-4	2	-2	-6
3	2	-2	-6	0	-4	-8	-2	-6	-10	1	-3	-7	-1	-5	-9	-3	-7	-11	0	-4	-8	-2	-6	-10	-4	-8	-12
4	6	2	-2	4	0	-4	2	-2	-6	5	1	-3	3	-1	-5	1	-3	-7	4	0	-4	2	-2	-6	0	-4	-8
5	10	6	2	8	4	0	6	2	-2	9	5	1	7	3	-1	5	1	-3	8	4	0	6	2	-2	4	0	-4
6	4	0	-4	2	-2	-6	0	-4	-8	3	-1	-5	1	-3	-7	-1	-5	-9	2	-2	-6	0	-4	-8	-2	-6	-10
7	8	4	0	6	2	-2	4	0	-4	7	3	-1	5	1	-3	3	-1	-5	6	2	-2	4	0	-4	2	-2	-6
8	12	8	4	10	6	2	8	4	0	11	7	3	9	5	1	7	3	-1	10	6	2	8	4	0	6	2	-2
9	1	-3	-7	-1	-5	-9	-3	-7	-11	0	-4	-8	-2	-6	-10	-4	-8	-12	-1	-5	-9	-3	-7	-11	-5	-9	-13
10	5	1	-3	3	-1	-5	1	-3	-7	4	0	-4	2	-2	-6	0	-4	-8	3	-1	-5	1	-3	-7	-1	-5	-9
11	9	5	1	7	3	-1	5	1	-3	8	4	0	6	2	-2	4	0	-4	7	3	-1	5	1	-3	3	-1	-5
12	3	-1	-5	1	-3	-7	-1	-5	-9	2	-2	-6	0	-4	-8	-2	-6	-10	1	-3	-7	-1	-5	-9	-3	-7	-11
13	7	3	-1	5	1	-3	3	-1	-5	6	2	-2	4	0	-4	2	-2	-6	5	1	-3	3	-1	-5	1	-3	-7
14	11	7	3	9	5	1	7	3	-1	10	6	2	8	4	0	6	2	-2	9	5	1	7	3	-1	5	1	-3
15	5	1	-3	3	-1	-5	1	-3	-7	4	0	-4	2	-2	-6	0	-4	-8	3	-1	-5	1	-3	-7	-1	-5	-9
16	9	5	1	7	3	-1	5	1	-3	8	4	0	6	2	-2	4	0	-4	7	3	-1	5	1	-3	3	-1	-5
17	13	9	5	11	7	3	9	5	1	12	8	4	10	6	2	8	4	0	11	7	3	9	5	1	7	3	-1
18	2	-2	-6	0	-4	-8	-2	-6	-10	1	-3	-7	-1	-5	-9	-3	-7	-11	0	-4	-8	-2	-6	-10	-4	-8	-12
19	6	2	-2	4	0	-4	2	-2	-6	5	1	-3	3	-1	-5	1	-3	-7	4	0	-4	2	-2	-6	0	-4	-8
20	10	6	2	8	4	0	6	2	-2	9	5	1	7	3	-1	5	1	-3	8	4	0	6	2	-2	4	0	-4
21	4	0	-4	2	-2	-6	0	-4	-8	3	-1	-5	1	-3	-7	-1	-5	-9	2	-2	-6	0	-4	-8	-2	-6	-10
22	8	4	0	6	2	-2	4	0	-4	7	3	-1	5	1	-3	3	-1	-5	6	2	-2	4	0	-4	2	-2	-6
23	12	8	4	10	6	2	8	4	0	11	7	3	9	5	1	7	3	-1	10	6	2	8	4	0	6	2	-2
24	6	2	-2	4	0	-4	2	-2	-6	5	1	-3	3	-1	-5	1	-3	-7	4	0	-4	2	-2	-6	0	-4	-8
25	10	6	2	8	4	0	6	2	-2	9	5	1	7	3	-1	5	1	-3	8	4	0	6	2	-2	4	0	-4
26	14	10	6	12	8	4	10	6	2	13	9	5	11	7	3	9	5	1	12	8	4	10	6	2	8	4	0

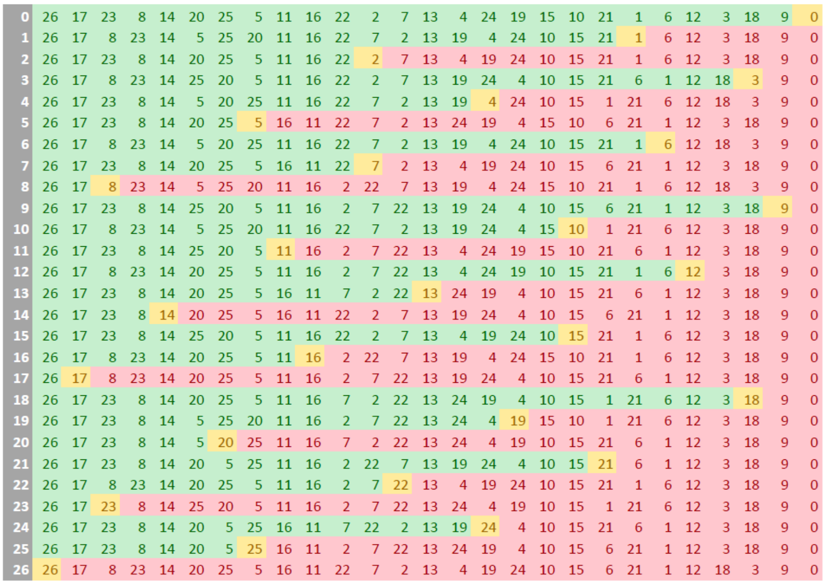
Figure 4.15: Request Receive matrix displaying ratings between a requested block group and a delivered block group

Figure 4.15 is an example of a 'request receive' matrix. The columns represent the type of block group requested and the rows represent the block group received. The value in the matrix represents how suitable the match between the block groups are where zero implies perfect match, positive numbers imply better than required and negative worse than required. For example in the sample matrix if I request a 12 and receive a 23 I get a benefit of +9. To calculate this matrix you first need to choose how much you value performance over reliability over space efficiency by choosing three integer constants. The constants in the example above are as follows: *Performance: 1, Reliability: 4, Space Efficiency: 2.* Figure 4.16 describes the method of calculating a rating.

$$\begin{aligned}
&ReceiveBlockType - RequestBlockType = \\
&\quad k_P \times (P_{Receive} - P_{Request}) + \\
&\quad k_R \times (R_{Receive} - R_{Request}) + \\
&\quad k_S \times (S_{Receive} - S_{Request})
\end{aligned}$$

Figure 4.16: Demonstrates how to calculate the trade off between block groups where P=performance, R=Reliability and S=Space Efficiency

This matrix can now be transformed into a traversal order of classifications. For example if a particular stream perfectly matches attribute 5 what other attributes would suite that stream and in what order should they be traversed? Using the example of attribute 5 the matrix in figure 4.17 suggests an ordering of[26,17,23,8,14,20,25....] where classifications nearer the beginning of the list are more suitable.



0	26	17	23	8	14	20	25	5	11	16	22	2	7	13	4	24	19	15	10	21	1	6	12	3	18	9	0
1	26	17	8	23	14	5	25	20	11	16	22	7	2	13	19	4	24	10	15	21	1	6	12	3	18	9	0
2	26	17	23	8	14	20	25	5	11	16	22	2	7	13	4	19	24	10	15	21	1	6	12	3	18	9	0
3	26	17	8	23	14	25	20	5	11	16	22	2	7	13	19	24	4	10	15	21	6	1	12	18	3	9	0
4	26	17	23	8	14	5	20	25	11	16	22	7	2	13	19	4	24	10	15	1	21	6	12	18	3	9	0
5	26	17	23	8	14	20	25	5	16	11	22	7	2	13	24	19	4	15	10	6	21	1	12	3	18	9	0
6	26	17	8	23	14	5	20	25	11	16	22	7	2	13	19	4	24	10	15	21	1	6	12	18	3	9	0
7	26	17	23	8	14	20	25	5	16	11	22	7	2	13	4	19	24	10	15	6	21	1	12	3	18	9	0
8	26	17	8	23	14	5	25	20	11	16	2	22	7	13	19	4	24	15	10	21	1	6	12	18	3	9	0
9	26	17	23	8	14	25	20	5	11	16	2	7	22	13	19	24	4	10	15	6	21	1	12	3	18	9	0
10	26	17	8	23	14	5	25	20	11	16	22	7	2	13	19	24	4	15	10	1	21	6	12	3	18	9	0
11	26	17	23	8	14	25	20	5	11	16	2	7	22	13	4	24	19	15	10	21	6	1	12	3	18	9	0
12	26	17	8	23	14	20	25	5	11	16	2	7	22	13	4	24	19	10	15	21	1	6	12	3	18	9	0
13	26	17	23	8	14	20	25	5	16	11	7	2	22	13	24	19	4	10	15	21	6	1	12	3	18	9	0
14	26	17	23	8	14	20	25	5	16	11	22	2	7	13	19	24	4	10	15	6	21	1	12	3	18	9	0
15	26	17	23	8	14	25	20	5	11	16	22	2	7	13	4	19	24	10	15	21	1	6	12	3	18	9	0
16	26	17	8	23	14	20	25	5	11	16	2	22	7	13	19	4	24	15	10	21	1	6	12	3	18	9	0
17	26	17	8	23	14	20	25	5	11	16	2	7	22	13	19	24	4	10	15	21	6	1	12	3	18	9	0
18	26	17	23	8	14	20	25	5	11	16	7	2	22	13	24	19	4	10	15	1	21	6	12	3	18	9	0
19	26	17	23	8	14	5	25	20	11	16	2	7	22	13	24	4	19	15	10	1	21	6	12	3	18	9	0
20	26	17	23	8	14	5	20	25	11	16	7	2	22	13	24	4	19	10	15	21	6	1	12	3	18	9	0
21	26	17	23	8	14	20	5	25	11	16	2	22	7	13	19	24	4	10	15	21	6	1	12	3	18	9	0
22	26	17	8	23	14	20	25	5	11	16	2	7	22	13	4	19	24	10	15	21	1	6	12	3	18	9	0
23	26	17	23	8	14	25	20	5	11	16	2	7	22	13	24	4	19	10	15	1	21	6	12	3	18	9	0
24	26	17	23	8	14	20	5	25	16	11	7	22	2	13	19	24	4	10	15	21	6	1	12	3	18	9	0
25	26	17	23	8	14	20	5	25	16	11	2	7	22	13	24	19	4	10	15	6	21	1	12	3	18	9	0
26	26	17	8	23	14	20	25	5	16	11	22	7	2	13	4	19	24	10	15	6	21	1	12	18	3	9	0

Figure 4.17: Matrix showing the order in which classifications should be traversed depending on what classification is required

Since I now have a classification for each block group, and a traversal order of classifications it is trivial to convert these into a traversal order of block groups on a per stream basis. The system can then traverse the pages in this order until it finds some free space.

4.4.2 Migration

Defragmentation

When it comes to storage systems it is expected in certain situations that the free space will grow and shrinks as time goes on. If the system gets to a high capacity and then shrinks back to a low capacity the data will most likely be scattered around the storage.

This is detrimental to performance for two reasons. If like data is spread around the storage then a lot of time will be wasted doing seek operations. The second and more important reason is that the older data would have been placed in less satisfactory locations since at the point where they were allocated space the more suitable locations were already taken. However after the systems free space grows these suitable locations are more abundant and since they are not being used are reducing the possible throughput of the system.

```
for( Stream stream : streams ) {
        for( File file : stream.getFiles() ) {
                Space space = FS.getBestSpace(stream)
                if( space.rating() > file.rating() )
                        FS.migrate(file, space)
                else
                        FS.reclaim(space)
        }
}
```

Figure 4.18: Pseudo-code for migrating data to more suitable locations

By periodically running the defragmentation algorithm described in figure 4.18 these wasted block groups can be filled with data increasing the throughput of the system.

Changing of Stream Characteristics

Streams are not static entities; their characteristics can change with time. A data stream which mostly issues small sequential reads could with time change to a data stream which mostly issues large random writes. As the stream characteristics change so does its perspective of the underlying storage. This is a problem however since if the characteristics change then the data placed under the old characteristics will now be in unsuitable places stunting performance. By using the same algorithm described in figure 4.18 when the streams characteristics change this performance deficit can be mitigated.

Chapter 5

Visualisation

In order to show that the simulation is doing what it's supposed to do for both debugging purposes and demonstration purpose, I created a visualisation engine to show how the storage is manipulated with time and to show how performance varies. The aim of the visualisation is to display clearly to the user the interesting and important aspects of the Simulation. I wrote this in Java since it provides very comprehensive libraries for writing GUI's. The visualisation consists of 4 main views: Stream View, Storage View, Graph View and Log View.

It is important to consider the streams individually when visualising the data placement and migration. This is since each stream will favor different areas of the underlying storage differently depending on their QoS attributes and combine performance profiles.

There are two main aspects of a data stream which are important to convey in the stream view:

1. Where the data it produced resides

2. What the underlying profile looks like

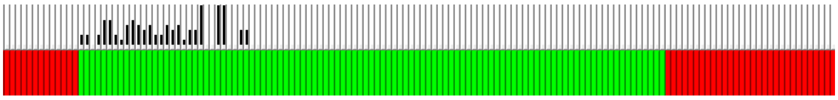


Figure 5.1: Diagram of Stream Animation

In figure 5.1 these two aspects are represented in the following way. The Upper strip indicates where data currently resides (A fully white bar meaning 'empty' and a fully black bar meaning 'full'). The lower strip represents how each area of the storage performs (Green – Better than required, Yellow – As Required, Red – Underperforms). In a simulation, multiple stream visualisations will be displayed in order to show how they interact with one another.

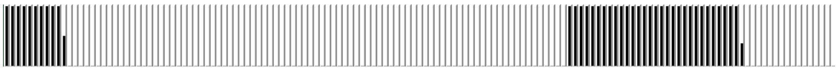


Figure 5.2: Diagram of Storage View

An additional panel known as the 'Storage View' is available which visualises the storage as a whole (stream independent). The storage view is similar to the stream view with the exception that is does not have the lower strip representing storage performance. The reasons for this view are two-fold, firstly to show a complete picture of what portion of the storage is occupied and secondly to show how the streams combine in the larger picture. This view is demonstrated in figure 5.2.

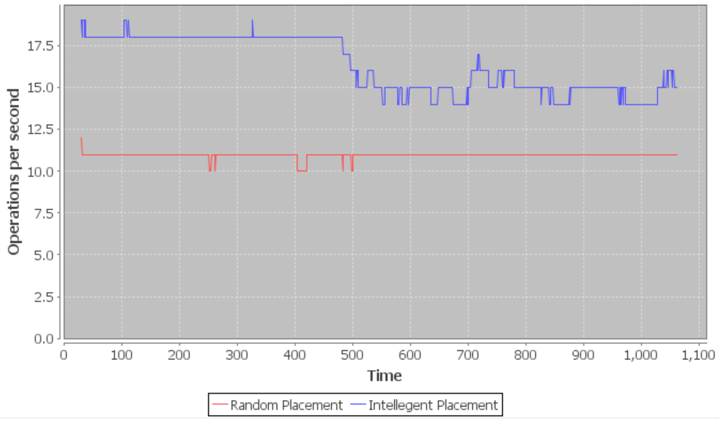
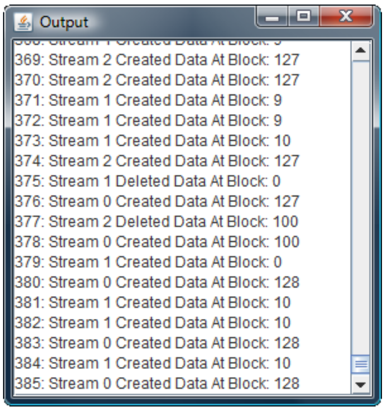


Figure 5.3: Diagram of Graphing View

The graph view[1] plots two datasets in real time. A rate graph of the intelligent placement technique and a rate graph of a random placement

[1] The graphing software used is an open source library called JFreeChart – `http://www.jfree.org/jfreechart/`

technique. This is used to show the performance benefit of using the intelligent placement technique. This view is demonstrated in figure 5.3.



```
369: Stream 2 Created Data At Block: 127
370: Stream 2 Created Data At Block: 127
371: Stream 1 Created Data At Block: 9
372: Stream 1 Created Data At Block: 9
373: Stream 1 Created Data At Block: 10
374: Stream 2 Created Data At Block: 127
375: Stream 1 Deleted Data At Block: 0
376: Stream 0 Created Data At Block: 127
377: Stream 2 Deleted Data At Block: 100
378: Stream 0 Created Data At Block: 100
379: Stream 1 Created Data At Block: 0
380: Stream 0 Created Data At Block: 128
381: Stream 1 Created Data At Block: 10
382: Stream 1 Created Data At Block: 10
383: Stream 0 Created Data At Block: 128
384: Stream 1 Created Data At Block: 10
385: Stream 0 Created Data At Block: 128
```

Figure 5.4: Diagram of Storage View

The final view is Log view. This view will display textually every operation performed on the storage ie ‘Create a file at block group 46’. This view is demonstrated in figure 5.4.

Chapter 6

Evaluation

For the purposes of this evaluation I will demonstrate my software using a series of case studies. The case studies will consist of running the simulation under different circumstances and inspecting the results using the animation.

For these tests I have used: *Dual Core 2 CPU (6600 @ 2.40GHz), 4096KB cache, 3.46 GB main memory, Ubuntu Linux 2.6.27-14-generic.*

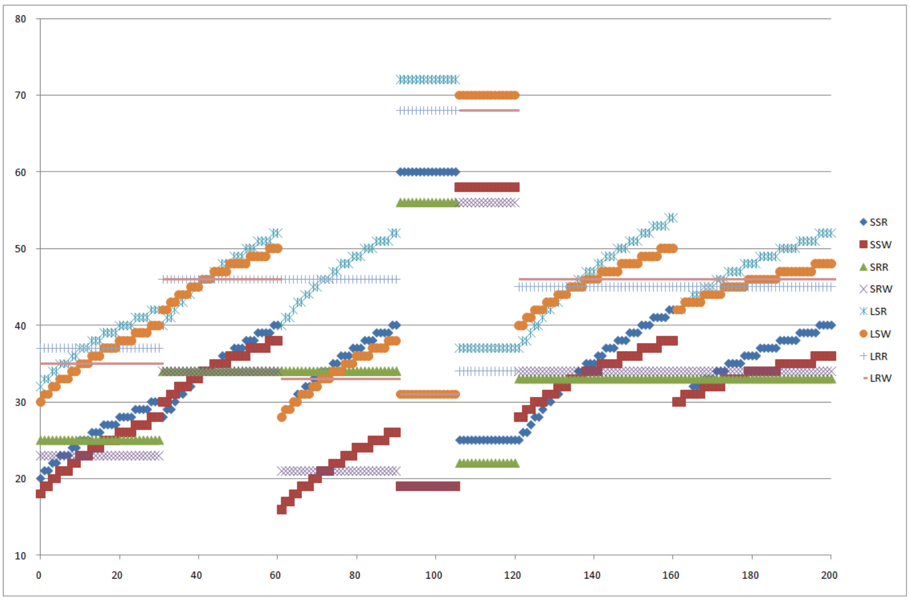


Figure 6.1: Graphical representation of the 8 profiles used for the case studies. The graph plots LBA against throughput in bytes/second.

Since each simulation requires an underlying performance profile I will be using a self fabricated profile in order to try and show some interesting characteristics of the techniques. Figure 6.1 is intended to simulate an LVM consisting of the following components:

1. Standard Magnetic Hard Disk
2. Raid 0 Array
3. Raid 1 Array
4. Read Biased Flash Memory
5. Write Biased Flash Memory
6. Raid 5 Array
7. Raid 10 Array

Each case study will be using different simulation settings. I will use the following notation to describe these settings: {*[1,H,M,L,50%,50%,50%]*}. This particular example would represent a simulation with a single data stream with characteristics: Proportion of 1, High Performance requirements, Medium Reliability requirements and Low Space Efficiency requirements. With equal part Small Sequential Reads to Large Random Writes. I will consistently use the QoS constants Performance: 2, Reliability: 3 and Space Efficiency: 1 for every case study. Videos of all the case studies can be viewed at `http://www.doc.ic.ac.uk/~hab06`.

6.1 Case Study: Order of placement

In order to demonstrate the order in which the simulation places data I initialised the simulation with the following settings: {*[1,M,M,M,50%,50%,50%]*}. When running the simulation log through the animation engine there are some clear results.

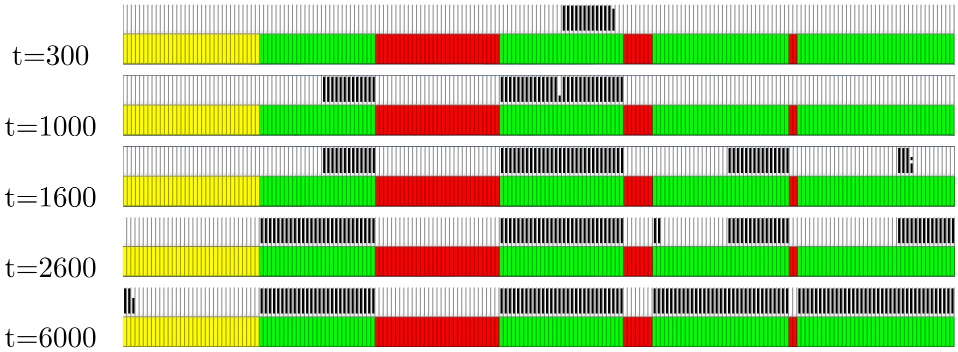


Figure 6.2: Sequential Animation Timeline Showing Correct Placement Ordering

Correctness of placement

As shown in figure 6.2 the green (better than required) portions of the storage are the first to be filled. This is because under these particular simulation settings these areas are most suitable for storing the data. Next (at t=6000) the yellow (as required) portions of the storage start to be filled since there are no more green areas. In this example since the simulation was only asked to fill 80% of the space the red portions of the storage will never be used since no more space is needed. This is an ideal situation since all the data will be 'happy' with where it is placed.

Correctness of Event Generation

Also shown by this case study is that the event generation is working correctly. When the simulation begins it is at 0% capacity and the number of creation events occurring are high since the probability of a creation is high (see section 4.3). These creation events occur less frequently as the storage approaches 80% capacity. This is expected since this example of the simulation was supposed to hover around 80% capacity.

6.2 Case Study: Performance Comparison

A major reason for using this method of allocation is to utilize the underlying storage in the most effective way to get the highest performance. The next experiment therefore is to measure the performance of the placement method in terms of rate of operations with respect to time. In order to get an absolute measurement I will also investigate the case where instead of using the intelligent placement technique I will use random placement. I will explore two scenarios with the same settings $\{[1,M,M,M,50\%,50\%,50\%]\}$:

Scenario A – Single stream, same settings as above, 80% free space aim

Scenario B – Single stream, same settings as above, 20% free space aim

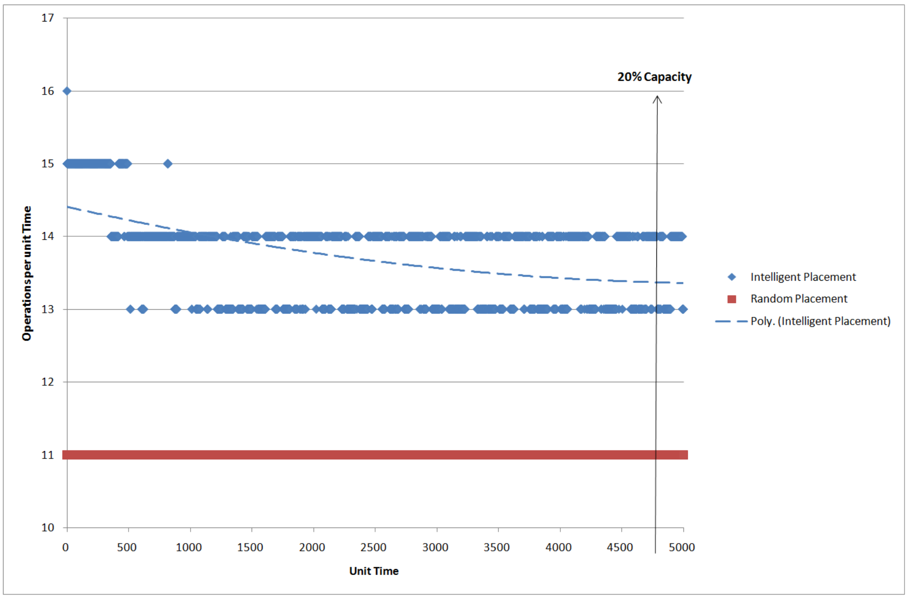


Figure 6.3: Diagram of Case Study Profile at a Low Capacity (Scenario A)

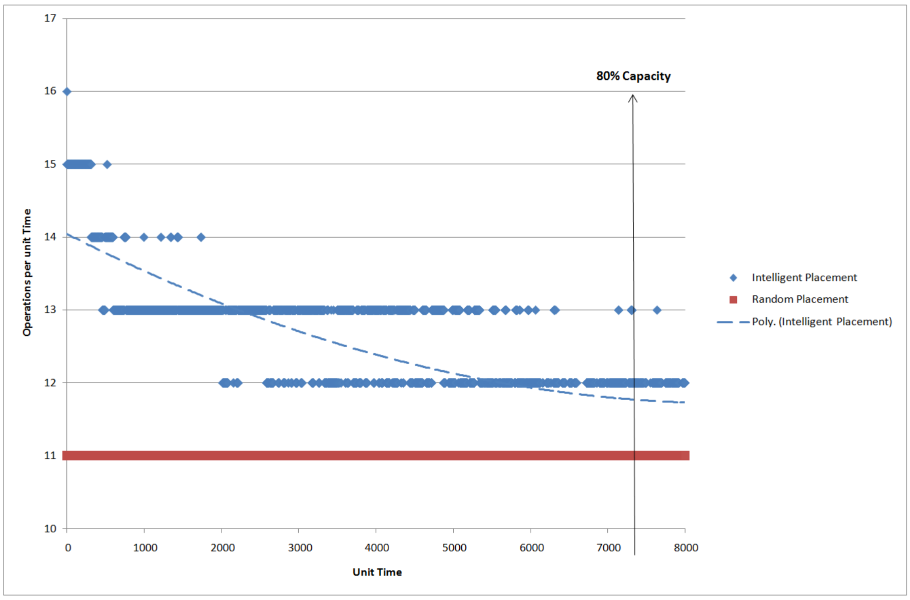


Figure 6.4: Diagram of Case Study Profile at a High Capacity (Scenario B)

The graph for scenario A (figure 6.3) demonstrates that by using the intelligent placement technique over the random placement technique there is a significant performance increase of nearly 2.5 disk operation per unit time. Scenario B (figure 6.4) however only offers a performance increase of 0.75 disk operations per unit time.

The reason for these results is as follows. When the system is kept at 20% capacity the probability of finding high performance areas of storage is high since they would not yet have been filled. This means that the majority of the data will be placed in these areas yielding very good performance for the system as a whole. When the capacity is set at 80% however the chance of finding a high performance area is far lower since they have been used to store other data. This leads to a performace deficit since the system is forced to use areas of lower performance. If the system capacity rose to 100% it would be expected that the intelligent placement curve would converge to the random placement line.

6.3 Case Study: Multiple Streams

Since in reality many data streams will be competing with each other to allocate space this next case study explores multiple concurrent streams. Figure 6.5 shows the animation of a run with 3 streams specified. The stream settings are as follows:
{*[1,H,M,L,17%,18%,84%], [3,L,M,H,78%,81%,13%], [1,M,M,M,50%,50%,50%]*}

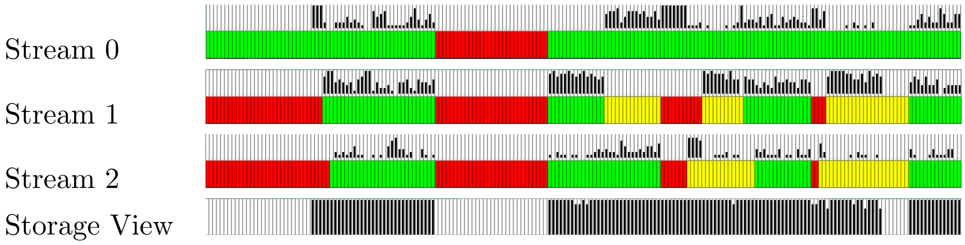


Figure 6.5: Multiple streams acting on the same storage mid simulation

Figure 6.5 demonstrates the multiple stream simulation. An interesting thing to notice about this run of the simulation is that since each stream was given very different characteristics the way each values its underlying storage is also very different. There are areas of Stream 1 which are marked red (unsuitable) which in Stream 0 are marked green (better than required). A result of this is that each stream will allocate different areas of the storage.

With this said it usually turns out that certain areas of the storage are preferred by all streams. This can lead to conflict between the streams, Stream 1 and Stream 2 clash in multiple areas where they are both trying to allocate storage. Since Stream 1 has a far higher proportion than Stream 2, Stream 2's data becomes quite scattered and fragmented. This could lead to performance problems depending on how Stream 2's data is accessed (see section 7.2/*Provisional Space Allocation*).

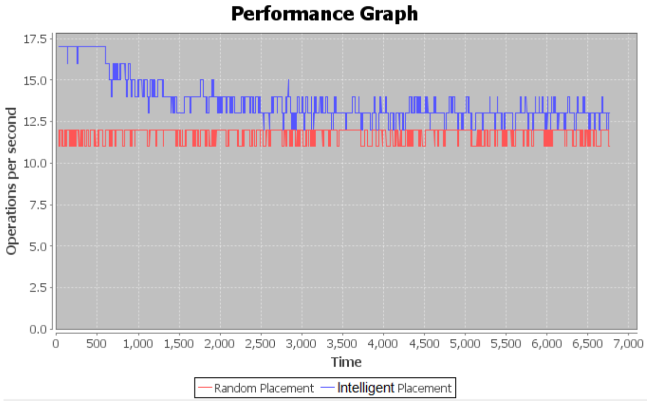


Figure 6.6: Performance Profile when multiple streams are acting on the same storage

Figure 6.6 shows the performance of the system over time. It turns out that when multiple streams are specified the performance increase is more prominent since at points the intelligent placement method is performing 2.5 operations per second more than the random method even at 80% capacity. The reason being that when more streams are specified the system specialises each stream's placement technique giving major performance benefits.

6.4 Case Study: Changing Stream Characteristics

In practice although data streams will have observable access patterns these patterns are likely to change with time. This could pose a problem since if for example the old access pattern expected a majority of reads and the stream changed to expect a majority of writes then the probability of the placement would not be optimal for the new characteristics. There is therefore a need to detect changing characteristics and to migrate data to accommodate these changes. The settings used for this case study are: $\{$*[1,L,M,H,78%,81%,13%]*$\}$.

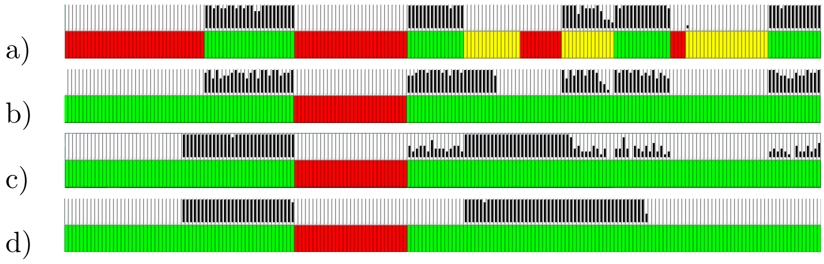


Figure 6.7: Example of changing QoS stream characteristics at different point of time

Figure 6.7 shows an example of a simulation where there is a change in a streams characteristics mid run.

- Snapshot a) A small time period before the characteristics of the stream change, where some data has already been placed in the most appropriate locations for this profile.
- Snapshot b) The point at which the streams characteristics have changed. Notice that portions of the storage which were deemed unsuitable before are now classified as good for the new characterstics.
- Snapshot c) The simulation in a mid migration stage where the data is being moved around into the most suitable areas for the new profile.
- Snapshot d) The final resting place of the data post migration.

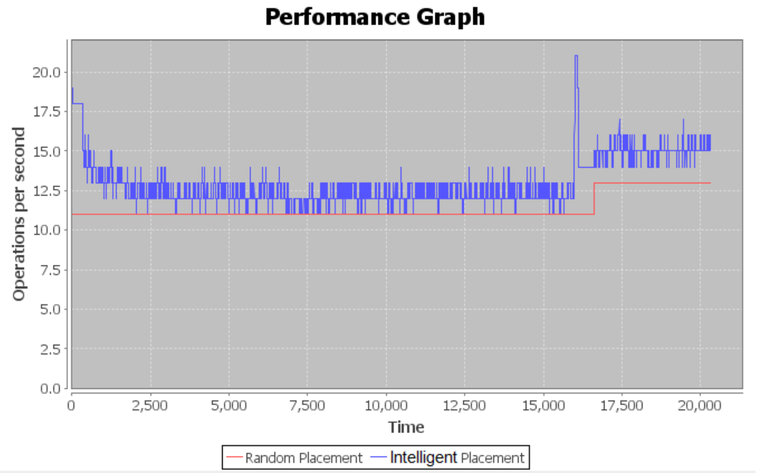


Figure 6.8: Performance Graph after a stream's characteristics have been changed mid simulation

Figure 6.8 shows a performance graph of this run of the simulation. The most noticeable aspect of the graph is the spike at t=16,000. This spike is caused by the migration code which writes the poorly placed data to the best available space. This space is very high performance hence the spike. Interestingly the stream's new performance profile yields better performance since it is more suited to the storage. If the migration code had not been run the performance would have tended far closer to the random placement line.

6.5 Case Study: Growing and Shrinking File System

It is to be expected that with time a system's capacity will grow and shrink. If a system grows to a large size then some of the newer pieces of data will be forced to reside in under performing areas due to lack of high performance areas. If the system were to then shrink to a smaller size there would be two things to observe:

1. High performance areas would become available

2. Some data would still remain in low performance areas

This is of course not utilizing the full potential of the system since the data in the low performing areas would be 'happier' in the free high performing areas. Migration techniques can be used to move this data into more appropriate locations. This next case study shows how migration techniques can be applied to mitigate this problem. Figure 6.9 shows an example of the simulation with settings {*[1,M,M,M,50%,50%,50%]*} as it goes through its various stages.

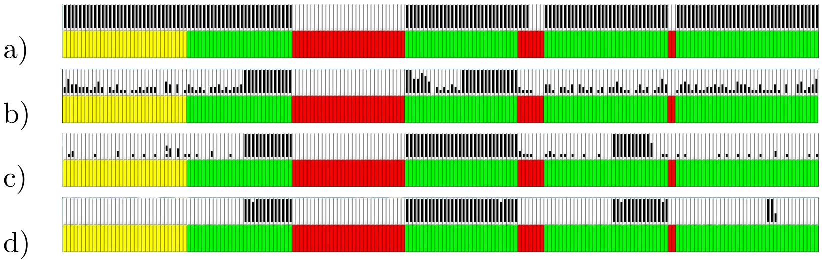


Figure 6.9: Example of a simulation when the file system grows and then shrinks at different time periods

- Snapshot a) The file system reaches its highest capacity.
- Snapshot b) The file system is approaching its lowest capacity.
- Snapshot c) The migration techniques are being applied and are 50% complete
- Snapshot d) The final resting place of the data post migration.

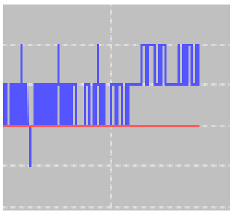


Figure 6.10: Performance Graph at the point just before and after defragmation migration

The performance of the system rises by an entire operation per unit time after the migration has been applied demonstrated in figure 6.10.

Chapter 7

Conclusion

7.1 Summary

This report has described methods of profiling and characterising a storage device under different conditions to achieve an indepth picture of how the device will perform. Algorithms have been described for allocating and migrating data on such a storage device to maximise performance, reliability and space efficiency. These algorithms have been put to practice using a simulation where data streams can be specified to operate on the storage. Lastly described is a visualisation technique for observing how data gets placed and migrated thoughout the simulation.

After observing results from the case studies in chapter 6 it seems obvious that there are major benefits of using these techniques. These benefits include a performance boost, increase in data reliability and cost saving with space efficiency.

A user should not be expected to know where their data should physically reside. Instead a user should be able to set some expectations of their data and have the system allocate the data as it sees fit.

There are many problems which my simulation does not cover which would need to be carefully considered if put into practice. One attractive feature of VSS (particularly linux's LVM) is the ability to add new devices and extend the file system dynamically. Adding a new device however would completely change the performance profile of the entire device. Hard drives are also prone to failure, if a hard drive were to fail this would also change the performance profile. For these reasons it is important for an implementation to periodically profile the VSS so it does not have an incorrect model.

Profiling itself has its own problems. Profiling takes a very long time to complete since is does relativly small operations to the entire device and 'eats' up the device's bandwidth. Since some of these operations will be writes it is important that when doing these writes no data is destroyed.

Data streams in my simulation were specified from the offset and could be changed mid run. In a real application of the system only the QoS attributes would be able to be specified by the user. The other attribues (i.e. the ratio small:large io) would be worked out by taking statistical information about the streams on the fly.

Taking all these considerations into account there is a possibility that the overheads would outweigh the benefits. However this is only speculation and can only be confirmed through further investigation.

7.2 Further Work

Since the time period allocated for this project was relatively short there are many aspects which I either didn't have time to explore or only thought about during implementation.

7.2.1 Load Balancing

Since this system would be intended to be run on VSSs there will be many separate storage devices involved. Usually the allocation of space would be assigned quite contiguously (i.e. one after another on the same device). Since it is using only one device at a time it is not taking advantage of possible parallelism between the devices (a pseudo raid 0 type striping scheme). This parallelism is a big motivation for spreading frequently accessed data around different storage devices as it would increase throughput greatly.

7.2.2 Simulated Drive Failure and Degraded Mode

One big problem of magnetic hard disks is that they are prone to failure. My implementation tries to mitigate this problem by placing more important data in more reliable sections of storage. What would be interesting is to simulate a drive failing and seeing what data gets affected. Another interesting experiment would be to have a single drive from a raid array fail and see how performance is affected in degraded mode, and how data would migrate away from the array until the drive is replaced.

7.2.3 Provisional Space Allocation

Since some areas of storage happen to be the most appropriate areas for data no matter what stream characteristics are set, streams often end up competing for the same space in some areas. This leads to the data being interleaved and spread for all the different streams. Since it is likely that the access of these data items will be sequential it is not optimal if it is spread across the device. For this reason there is motivation for allocating provisional areas for each data stream. This would lead to far less stream wise fragmentation and a higher throughput.

7.3 Closing Remarks

Overall I believe this project has been successful in showing that by using intelligent data placement techniques and a performance profile a systems throughput and reliability can be greatly increased. My experiments show that such a system is viable and I hope that future work will build upon these methods and come up with a concrete implementation.

Bibliography

[1] Medha Bhadkamkar, Jorge Guerra, Luis Useche, Sam Burnett, Jason Liptak and Raju Rangaswami *BORG: Block-reORGanization and Self-optimization in Storage Systems* (2007)

[2] Felipe Franciosi *MPhil to PhD Transfer Report (Imperial College London): Data Placement and Migration in Virtualised Storage Systems* (2008)

[3] Daniel Bovet, Marco Cesati *Understanding the Linux Kernel, Third Edition* (2008)

[4] Ram Kumar, Rakesh Agrawal *Programming in ANSI C* (1992)

[5] Lee, E.K.: Performance Modeling and Analysis of Disk Arrays. PhD thesis, University of California at Berkeley (1993)

[6] Chen, S., Towsley, D.: The design and evaluation of RAID 5 and parity striping disk array architectures. IEEE Transactions on Parallel and Distributed Systems 17 (1993)

[7] S. Chen and D. Towsley. A performance evaluation of RAID architectures. IEEE Transactions on Computers, (1996)

[8] A. S. Lebrecht and W. J. Knottenbelt. Response time approximations in fork-join queues, July (2007)

[9] S. Zertal and P. G. Harrison. Multi-RAID queueing model with zoned disks, June (2007)

[10] Gantz, J.F.: The expanding digital universe: A forecast of worldwide information growth through 2010. White paper, IDC (2007)

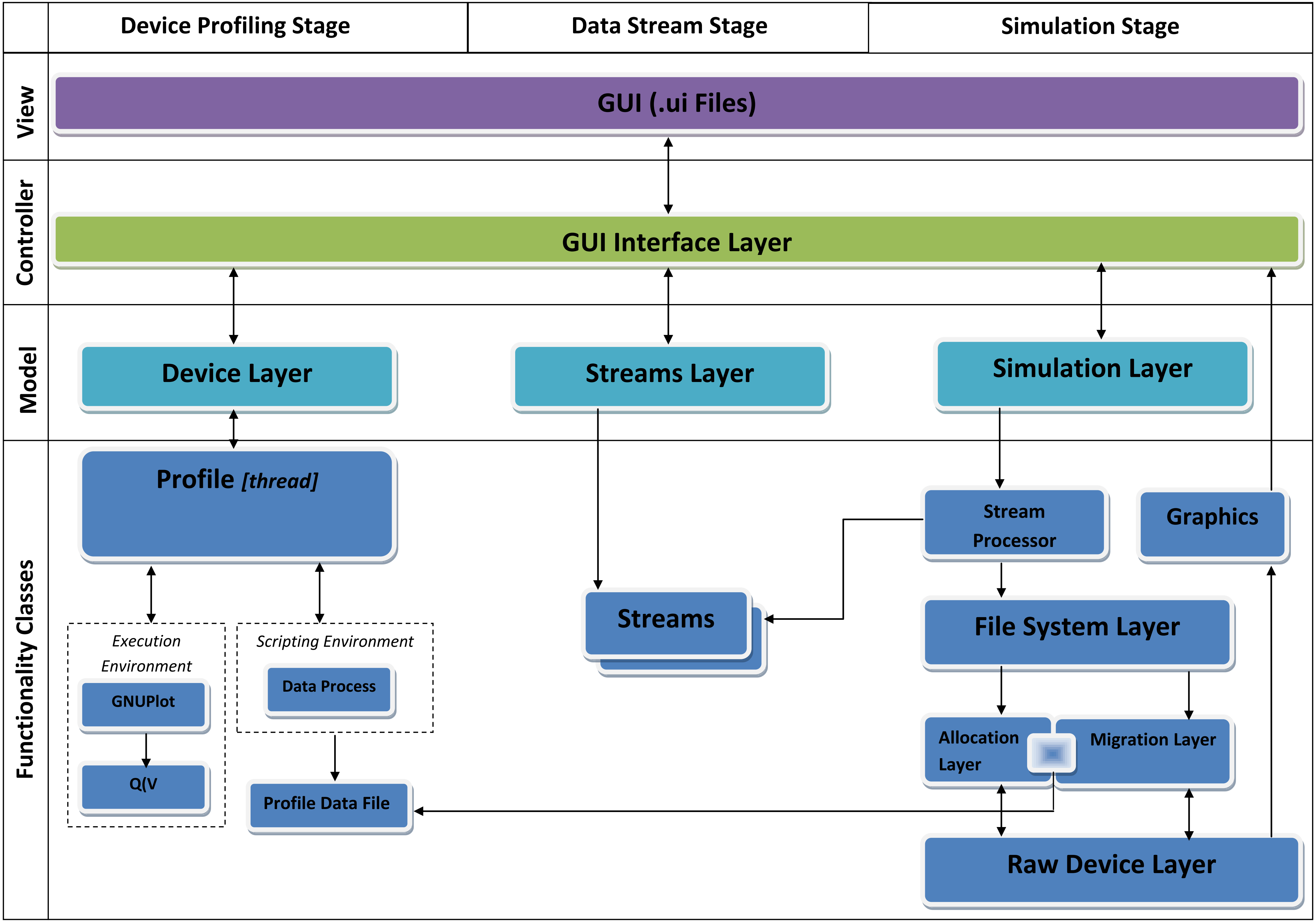
[11] Wikipedia RAID *http://en.wikipedia.org/wiki/Redundant_array_of_independent_disks* (2008)

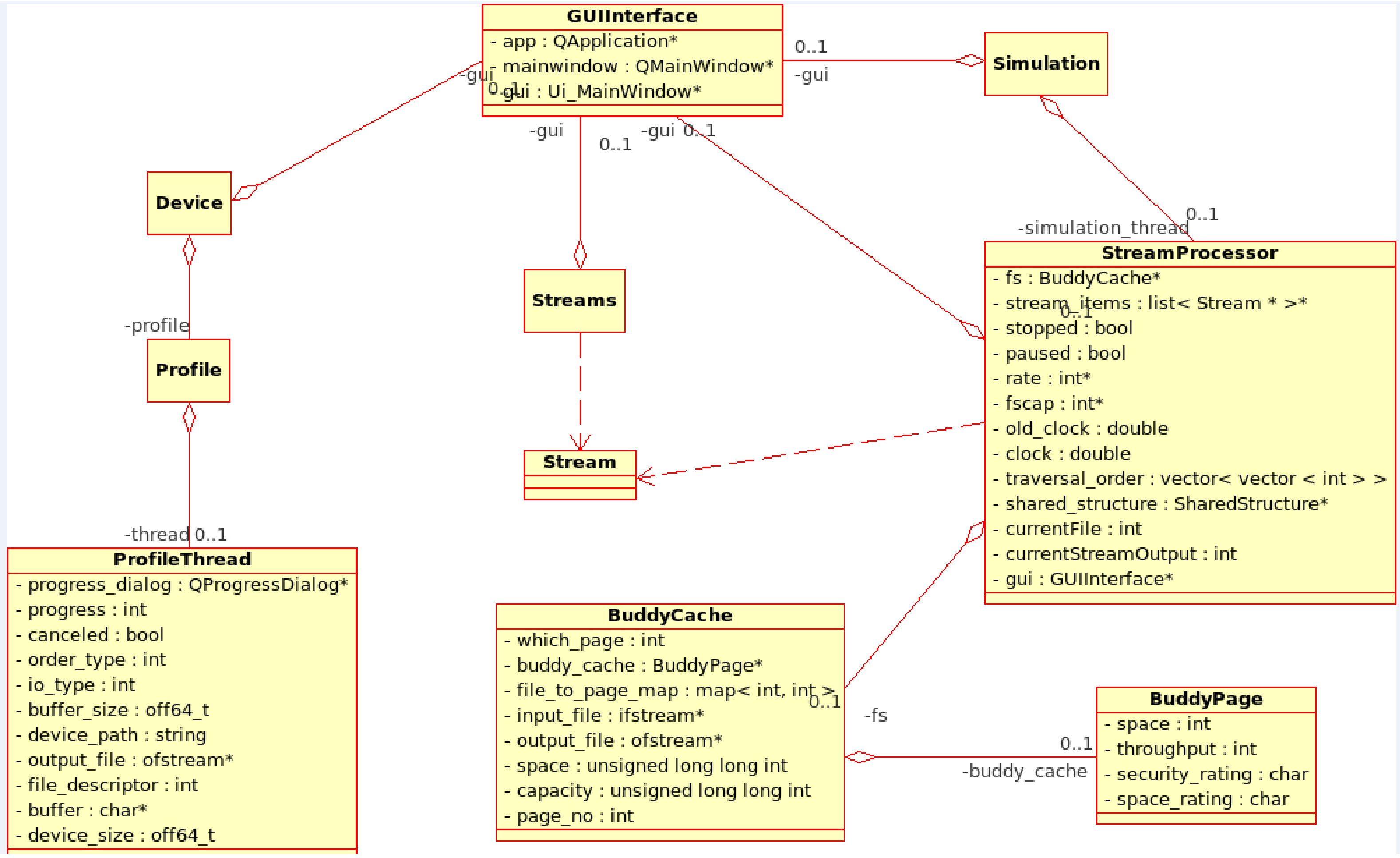
[12] Wikipedia Hard Disk Drive *http://en.wikipedia.org/wiki/Hard_drive* (2008)

List of Figures

Appendix A

Design Diagrams


Device Profiling Stage
Data Stream Stage
Simulation Stage
View
Controller
Model
Functionality Classes
GUI (.ui Files)
GUI Interface Layer
Device Layer
Streams Layer
Simulation Layer
Profile [thread]
Execution Environment
GNUPlot
Q(V
Scripting Environment
Data Process
Profile Data File
Streams
Stream Processor
Graphics
File System Layer
Allocation Layer
Migration Layer
Raw Device Layer


GUIInterface
- app : QApplication*
- mainwindow : QMainWindow*
- gui : Ui_MainWindow*
0..1
-gui
Simulation
-gui
0..1
-gui
0..1
-gui 0..1
Device
-profile
Profile
Streams
Stream
-simulation_thread
0..1
StreamProcessor
- fs : BuddyCache*
- stream_items : list< Stream * >*
0..1
- stopped : bool
- paused : bool
- rate : int*
- fscap : int*
- old_clock : double
- clock : double
- traversal_order : vector< vector < int > >
- shared_structure : SharedStructure*
- currentFile : int
- currentStreamOutput : int
- gui : GUIInterface*
-thread 0..1
ProfileThread
- progress_dialog : QProgressDialog*
- progress : int
- canceled : bool
- order_type : int
- io_type : int
- buffer_size : off64_t
- device_path : string
- output_file : ofstream*
- file_descriptor : int
- buffer : char*
- device_size : off64_t
BuddyCache
- which_page : int
- buddy_cache : BuddyPage*
- file_to_page_map : map< int, int >
0..1
-fs
- input_file : ifstream*
- output_file : ofstream*
- space : unsigned long long int
- capacity : unsigned long long int
- page_no : int
BuddyPage
- space : int
- throughput : int
- security_rating : char
- space_rating : char
0..1
-buddy_cache

Appendix B

Contact Details

If you have any questions about this project please feel free to contact me at: hab06@doc.ic.ac.uk