**Owain D Lloyd**

# Constant Transfer Rate RAID striping for QoS

Part II Computer Science

2002

Gonville & Caius College

# Proforma

Name : **Owain D Lloyd**
College : **Gonville & Caius**
Project Title : **Constant Transfer Rate RAID striping for QoS**
Examination : **Part II Computer Science, 2002**
Word Count : **8700**
Project Originator : Dave Stewart
Supervisor : Richard Watts


## Abstract

Currently the performance observed by users of disks is affected greatly by where the data is placed on that disk. Towards the outside of the disk there are a greater number of sectors that pass under the head quickly, so higher throughput is observed, than on the inside of the disk. For applications such as random access streaming media servers better load balancing is needed.

I propose a level of RAID striping that enforces a constant disk transfer rate to achieve near to average performance across the entire logical disk.


## Work Completed

Primarily, a patch the the Linux 2.4 series kernel tree to provide the personality of RAID striping developed in this document. Along with this is an extensive patch to the user space utilities package 'raidtools'. This satisfied the aim of the project and required the learning of C and how to program for the Linux Kernel.

Secondly, a patch to the kernel and user space character device driver to record low level SCSI traces (`ll_trace`).

In addition, this dissertation requiring the learning of LaTeX.

## Special Difficulties

The disk drives I had to work with turned out to have different internal structure despite the the same model numbers. This prevented work on one of the algorithms I developed.

# Declaration of Originality

I Owain D. Lloyd of Gonville and Caius College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed:

Date:

# Contents

**1 Introduction** **1**
1.1 Related Work . . . . . 1
1.2 Chapter Layout . . . . . 1

**2 Preparation** **3**
2.1 Requirements . . . . . 3
2.2 Choice of Tools . . . . . 3
2.3 Evaluation Methodology . . . . . 4
2.4 Simulation . . . . . 4
2.4.1 Synthetic workloads vs. Traces . . . . . 4
2.4.2 Pantheon and DiskSim . . . . . 5
2.5 The Development Environment . . . . . 6
2.6 Modern Disks . . . . . 6
2.6.1 Zoned bit recording . . . . . 6
2.6.2 Logical block addressing (LBA) . . . . . 7
2.6.3 Track and Cylinder Skew . . . . . 7
2.6.4 Defect Management . . . . . 7
2.6.5 Device Prefetching . . . . . 8
2.7 A development of possible solutions . . . . . 9
2.7.1 The Linux Multiple Devices (MD) Block Driver . . . . . 9
2.7.2 QoS RAID as an extension to the MD driver . . . . . 9
2.7.3 Geometry Issues . . . . . 10
2.7.4 Assumptions . . . . . 10
2.7.5 Reading backwards, Biasing and Pre-fetching . . . . . 11
2.7.6 `qmap_nogeo` . . . . . 11
2.7.7 Logical Zoning . . . . . 11
2.7.8 `qmap_geo` . . . . . 13
2.8 Media Streams . . . . . 13

**3 Implementation** **15**
3.1 Configuring DiskSim . . . . . 15
3.2 Recording of Simulation traces . . . . . 15
3.3 Application of mappings to simulation traces . . . . . 15
3.4 Download and Installation instructions . . . . . 16
3.5 The Kernel Patch . . . . . 16
3.6 The Raidtools Package . . . . . 18
3.7 The Low Level Disk Trace Logger . . . . . 19

**4 Evaluation** **21**
4.1 Results of Simulation . . . . . 22
4.2 Validation of the Mapping Functions . . . . . 24
4.3 Measurement of Implementation Statistics . . . . . 24
4.4 Implementation Results . . . . . 24
4.4.1 Testing a streaming video server: . . . . . 25

**5 Conclusions** **27**
5.1 Further Directions . . . . . 27
5.1.1 Specifying a variable QoS bound . . . . . 28
5.1.2 Extracting the PBN/LBN geometry mapping . . . . . 28
5.1.3 Hardware that reads backwards . . . . . 28

**A Low Level SCSI Trace Driver** **31**

**B QoS RAID kernel code** **33**

**C Raid Tools Code** **35**

**D Tracing the Media Server** **37**

# List of Figures


2.1 A Zoned Disk Surface . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
2.2 IBM Deskstar 40GV Zone Transfer Rates . . . . . . . . . . . . . . . . . . . 8
2.3 IBM Deskstar 34GXP Geometry . . . . . . . . . . . . . . . . . . . . . . . . 8
2.4 Logical zoning example . . . . . . . . . . . . . . . . . . . . . . . . . . . 12

3.1 The layout of source files in the Linux implementation . . . . . . . . . . . 17
3.2 A cache efficient array layout . . . . . . . . . . . . . . . . . . . . . . . . 17
3.3 An sample /etc/raidtab for Logical Zoning . . . . . . . . . . . . . . . . . . 18

4.1 Configuring a QoS RAID kernel . . . . . . . . . . . . . . . . . . . . . . . 21
4.2 Summary of Simulation Response Times . . . . . . . . . . . . . . . . . . . 22
4.3 Graph to show summary of Simulation response times . . . . . . . . . . . 23
4.4 Simulation Seek Times . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
4.5 Table of Request Completion Times . . . . . . . . . . . . . . . . . . . . . 24
4.6 Graph of Request Completion Time against location of data on disk . . . . 25

# Chapter 1

# Introduction

Modern disks use a technique called *Zoning* (See Section 2.6.1) to give better data density and faster transfer rates towards the outside of the disk. The unbalanced transfer rates across the disk mean only the slowest (inside) rate can be guaranteed.

The envisaged use of the work described in this document is for high bandwidth real time applications such as streaming (non-live) video servers where the transfer rate from the disk must be guaranteed to deliver a reliable set of streams at a particular bitrate. At present, a common approach to raise the minimum transfer rate is to only use the outside of the disks in an array. Disks are cheap but high bitrate video demands lots of storage as well as speed and QoS, rendering this wastage unacceptable.

Other possible uses include intensive log file and audit trail writes.

## 1.1 Related Work

As the concept of constant transfer rate striping is entirely new, I know of no existing work in this area. Much of the work done by John Wilkes' team in the Storage Systems Program at HP Labs [5] and Professor Greg Ganger's team at Carnegie Mellon's Parallel Data Lab [1] is in the broad area of measuring and improving disk performance.

Other Storage QoS related work includes a QoS scheduling and management system for large distributed storage systems in [15] and a data migration system to enforce a bound on foreground performance impact [9].

Many aspects of system performance can be measured and events logged by the Linux Trace Toolkit [19]. Unfortunately it does not log low level disk traces which led to my development of the LL_TRACE module discussed in section 3.7.

Ruemmler [12] analyzes many days recordings of disk traces from standard unix servers. However, they are not operating in the conditions QoS RAID is intended for; namely high load streaming media.

The Linux Raid mailing list [8] is used to release lots of work relating to the Linux RAID code and is the only documentation for the kernel RAID code.

## 1.2 Chapter Layout

In the next chapter I give some background to the problem, explaining the cause of variable transfer rates from hard disks and why that may be a problem in some types of servers.

I suggest various ways this could be solved using the concept of a striped RAID array, introducing the concept of *logical zoning*.

I evaluate two software simulators designed for experimenting with disk configurations. Results of running one of these simulators with a configuration to emulate the algorithms discussed in chapter 2 are presented in chapter 4.

Other research undertaken before coding started including an analysis of workloads is also listed in chapter 2. Chapter 3 presents the actual implementation for the Linux operating system with the results of testing in Chapter 4. A summary and a discussion of possible further work fills Chapter 5.

While the total line count for the kernel patch, the `raidtools` modification, the trace driver, the simulation scripts and the test code exceeds five thousand lines of concise C and Perl, I have included some of the most important code snippets in the Appendices.

## Acknowledgements

Thanks are due to Graham Titmus for helping to find a supervisor (and indeed to Richard Watts for his in depth knowledge of the kernel) and to Arjune Budhram for discussions in mathematics.

# Chapter 2

# Preparation

## 2.1 Requirements

The Implementation should be in the form of a Linux kernel patch. Supporting user space utilities should be provided to create and control the RAID array. Both should be documented in the standard form (Kernel documentation, README's and "man" pages). The kernel changes should be modular so changes can be unloaded to return to a vanilla kernel without re-compiling.

The patch should provide a new RAID personality that improves the minimum transfer rate from a disk array against using RAID 0 with identical hardware.

## 2.2 Choice of Tools

**Development Tools:** As the code was to be tested using the Linux console, I chose to use CVS for source control, vi for my editor and "screen" to enhance the terminal features. I am quick and comfortable in vi and "screen" but my limited knowledge of CVS needed some familiarization.

**Kernel Code:** There is no option but to use C for Kernel programming. Also the raidtools package that must be extended is already written in C. While I had three months experience in C++ coding, I had never used C and never programmed for the Linux Kernel. In August 2001 I read Rubini's [11] book on Linux Device Drivers, played with writing some sample code and familiarized myself with the Linux RAID code.

The Low Level Disk Trace driver gave me valuable experience before embarking on the RAID code.

**Scripting:** I used Bash for shell scripting as this is my default shell. The scripts for manipulating simulation data were written in Perl as it is fast to prototype in and powerful. I am comfortable in both.

**Documentation:** LaTeX , which had to be learned from scratch, was used for the dissertation and documentation was written as "man" pages.

## 2.3 Evaluation Methodology

How do we define success? Before embarking on any coding I considered what statistical measures could be used to compare a QoS Array with a RAID 0 array and a single disk. The fundamental requirement is that more bandwidth can be guaranteed from the disk array.

The most important measure is the **maximum request completion time** of a read to any part of the disk. The **standard deviation** of the distribution of request completion times across the disk is also useful. A small standard deviation shows the best utilization of the disk performance. The **mean** request completion time shows how the array performs if an upper bound is not important (e.g. for a root file system). This is not relevant to our requirements but interesting as it shows the overhead of performing a sector remapping and using this on-disk data layout.

A sensible test to run that would give a more practical feel for the performance gains would be to test a streaming server, counting how many feeds could be served before frames needed to be dropped.

## 2.4 Simulation

Kernel programming has many tricky facets. Amongst other problems, it is difficult to debug and requires a reboot after every change of the code. Fortunately there are some fully featured disk simulators already built that allow us to experiment with algorithms and obtain accurate results without the complications of kernel code.

To see whether our algorithms are actually worth implementing we should only need to:

1. Configure a simulator with two drives.
2. Collect a variety of disk access traces.
3. Write the sector remapping algorithms that we wish to simulate and a script that applies the mappings to the traces.
4. Run the simulator on the original traces and on the re-mapped traces, collecting the relevant results.

### 2.4.1 Synthetic workloads vs. Traces

"Traces" are files of disk access requests. Each entry consists of at least:

- Request arrival time.
- Device id.
- Block number.
- Request size.
- Type of request (Read or Write).

It would be very useful if we could use real traces, recorded on a system performing a relevant activity, e.g. serving multiple video streams, for our simulation. However it is rather difficult to record such a trace. Using a utility such as Linux's `strace(1)`, which records all system calls and can produce a file of all reads and writes during a process

execution, is inadequate. It will only record the kernel API functions (system calls) `read` and `write`, not the internal low level call `ll_rw_blk`; we are unable to record the actual block address.

To record low level access under load without affecting the timing we really need direct support in the kernel, or even better hardware connected to the disk controller.

Some simulators (such as DiskSim [4]) allow synthetic traces to be generated. Ganger [3] discusses how to generate realistic synthetic workloads. In fact Ganger is one of the designers of DiskSim, so his methods are used by the internal synthetic workload generator.

The problem with synthetic workloads is that the statistical measures of request arrival times used to produce them are not actually sufficient to describe a real situation [3]. In particular the only way to tune these generators to produce traces matching a given environment is to record real traces in the required environment and measure statistical properties, modeling the traces as random variable distributions. So either way, we need to record real traces. The only advantage of using a workload generator is that the size of the traces, and the intensity of access can be varied without recording more traces.

The extensive Linux Trace Toolkit [19] does not record Low Level traces either and the freely available traces described in [12] are not gathered in an appropriate environment (they are from standard unix servers from the mid 1990's not streaming media servers). So a dedicated low level trace recorder must be written too.

### 2.4.2 Pantheon and DiskSim

**Pantheon** is a storage-system simulator originally developed by Hewlett Packard in 1992. See Wilkes [14] for a full description. It is written in C++ so the compiled and linked executable runs at full speed. The configuration is written in the interpreted scripting language Tcl meaning extremely flexible parameters can be used. It comes with many modules for analyzing output and working with traces.

The reliability of Pantheon has been proved by several projects including HP AutoRAID [16].

**DiskSim** [4] is an efficient, accurate and highly-configurable disk system simulator developed at Carnegie Mellon University to support research into various aspects of storage subsystem architecture. It includes modules that simulate disks, intermediate controllers, buses, device drivers, request schedulers, disk block caches, and disk array data organizations. In particular, the disk drive module simulates modern disk drives in great detail and has been carefully validated against several production disks (with accuracy that exceeds any previously reported simulator). It can be either trace driven or use the internal synthetic workload generator. It is written in C so can be easily modified (after learning C!).

**My Choice:** I choose to use DiskSim as it is smaller and easier to work with. It is also the most accurate simulator of the two. In addition I do not need the advanced configuration or the trace analysis tools offered by Pantheon.

## 2.5 The Development Environment

A fresh VMWare 3 virtual machine running Linux 2.4.17 under Windows XP Professional was set up for the development and simulation work. The supporting box was a 1.1GHz Athlon with 1Gb RAM. Two extra virtual 200mb SCSI drives were provided by VMWare for RAID test use.

Two platforms were made available for performance testing.

1. A dual processor Sun HyperSPARC 150Mhz running SPARC Linux with 400Mb RAM and two 2GB 5400rpm IBM SCSI disk drives on an SBus SCSI-2 controller.
2. A uni-processor Athlon 1.1Ghz running Linux with 1Gb RAM and two 30Gb IBM Deskstar 75GXP 7200rpm 2mb cache IDE drives, each on a dedicated channel to a Highpoint UDMA/100 controller on an AMD 762 northbridge with 200Mhz DDR FSB.

The C code, scripts, simulator configurations and LaTeX sources were all stored in a CVS repository to control versioning and backtracking. The repository was backed up to both a CD-RW and the University of Cambridge *Pelican* backup server on a regular basis.

## 2.6 Modern Disks

I summarize in this section the specific information about disks that I needed to consider in the design of my algorithms.

### 2.6.1 Zoned bit recording

To eliminate wasted space in long outer sectors, modern hard disks employ a technique called *zoned bit recording (ZBR)* (*multiple zone recording* or *zone recording*). With this technique, tracks are grouped into zones based on their distance from the center of the disk, and each zone is assigned a number of sectors per track. As you move from the innermost part of the disk to the outer edge, you move through different zones, each containing more sectors per track than the one before. This allows for more efficient use of the larger tracks on the outside of the disk.

The side effect of this design is that the raw data transfer rate (*media transfer rate*) of the disk when reading the outside cylinders is much higher than when reading the inside ones. This is because the outer cylinders contain more data, but the angular velocity of the platters is constant regardless of which track is being read. Since hard disks are filled from the outside in, the fastest data transfer occurs when the drive is first used.

Figure 2.2 lists some real data for the 20 GB/platter, 5400 RPM IBM 40GV drive [6]. Only 173.8 Mbits/s can be guaranteed despite an average of 282 Mbits/s.

ZBR drives cannot have their true geometries expressed using three simple numbers (cylinders, heads, sectors). To get around this issue, for disks 8.4 GB or smaller, the OS is given bogus parameters that give the approximate capacity of the disk, and the hard disk controller is given intelligence so that it can do automatic translation between the logical and physical geometry. The actual physical geometry is totally different, but the BIOS (and the OS) need know nothing about this.

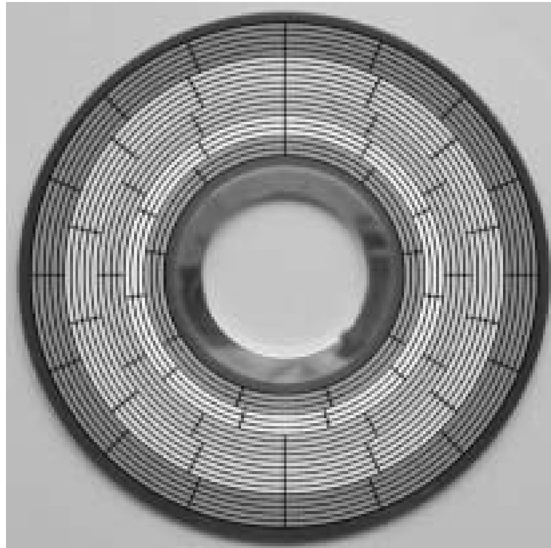

Figure 2.1: A Zoned Disk Surface

### 2.6.2 Logical block addressing (LBA)

Another way to get around the problem of complex internal geometry is to change the way the drive is addressed completely. Instead of using the logical geometry numbers directly, most modern drives can be accessed using logical block addressing (LBA). With this method a totally different form of logical "geometry" is used: the sectors are just given a numerical sequence starting with 0. Again, the drive internally translates these sequential numbers into physical sector locations.

Drives larger than 8.4 GB just specify 16,383 cylinders, 16 heads and 63 sectors to the BIOS for compatibility. Then, access to the drive is performed directly by the Int 13h extension routines, and the logical parameters are completely ignored. Figure 2.3 shows how a modern drive, the 34.2 GB IBM Deskstar 34GXP (model DPTA-373420), looks.

As you can see, the logical and physical geometries clearly have nothing to do with each other on drives this large, and even the total number of sectors is wrong in the logical geometry. Since they cannot have their true capacity expressed in terms of even logical geometry, all large modern drives are accessed using logical block addressing.

### 2.6.3 Track and Cylinder Skew

The first logical block of each track or cylinder is generally offset (skewed) from the first logical block of the previous track or cylinder to compensate for the positioning delay incurred when switching heads or seeking between adjacent cylinders. Thus a request that crosses a track or cylinder boundary does not suffer unnecessary rotational delay due to "just missing" the next logical block and waiting almost a full rotation for it to come around again.

### 2.6.4 Defect Management

Disk drives are designed to survive the loss of hundreds of sectors or tracks to media corruption. In fact, new drives usually contain a list of defects identified at the factory. Defects

| Zone | Tracks in Zone | Sectors per Track | Data Transfer Rate (MBits/s) |
|---|---|---|---|
| 0 | 624 | 792 | 372.0 |
| 1 | 1,424 | 780 | 366.4 |
| 2 | 1,680 | 760 | 357.0 |
| 3 | 1,616 | 740 | 347.6 |
| 4 | 2,752 | 720 | 338.2 |
| 5 | 2,880 | 680 | 319.4 |
| 6 | 1,904 | 660 | 310.0 |
| 7 | 2,384 | 630 | 295.9 |
| 8 | 3,328 | 600 | 281.8 |
| 9 | 4,432 | 540 | 253.6 |
| 10 | 4,528 | 480 | 225.5 |
| 11 | 2,192 | 440 | 206.7 |
| 12 | 1,600 | 420 | 197.3 |
| 13 | 1,168 | 400 | 187.9 |
| 14 | 1,815 | 370 | 173.8 |

Figure 2.2: IBM Deskstar 40GV Zone Transfer Rates

| Specification | Physical Geometry | Logical Geometry |
|---|---|---|
| **Read/Write Heads** | 10 | 16 |
| **Cylinders (Tracks per Surface)** | 17,494 | 16,383 |
| **Sectors Per Track** | 272 to 452 | 63 |
| **Total Sectors** | 66,835,440 | 16,514,064 |

Figure 2.3: IBM Deskstar 34GXP Geometry

grown during the course of the disk's lifetime are added to this list. Logical Block Number to Physical Block Number (LBN - to - PBN) mappings are adjusted so that defective areas are not used for data storage. Such adjustments fall under the categories of **slipping** and **reallocation**.

Sector or track slipping takes place when a disk is formatted. In the case of sector slipping, the formatting process skips each defective sector when initializing the LBN - to - PBN mapping. If a defective sector is in the "middle" of a track, the mapping is adjusted so that the sectors immediately adjacent to the defective sector will be logically sequential. Track slipping follows the same routine, except that entire tracks are skipped if they contain any defective media. Extra space is reserved at the end of certain tracks, cylinders, or zones to contain the spillover from the slipping process. If these spare regions are organized as a set of cylinders at the end of each zone, the seek delay will increase whenever a zone boundary must be crossed.

Reallocation occurs when defective sectors (or tracks) are discovered during normal disk usage. The affected LBN's are dynamically remapped to spare regions. Any subsequent accesses are redirected to the appropriate spare sectors.

### 2.6.5 Device Prefetching

On-board disk caches exploit spatial and temporal locality of reference in the same way as processor cache does for memory accesses. The disk automatically prefetches data into the cache to satisfy sequential read requests quicker. For example, a disk might take 20 ms to

service a read request that requires media access. A subsequent sequential read might take only 2 ms if the requested data blocks have been prefetched into the onboard disk cache (Worthington [17]).

## 2.7 A development of possible solutions

Throughout this section, as I develop algorithms to deliver constant transfer rates, I will use the the terms **Edge**, **Spindle** and **Midpoint** to mean the areas of the disk surface on the *Outer-most tracks (furthest from the center)*, the *Inner-most tracks* and the *tracks in the middle of the disk* respectively.

### 2.7.1 The Linux Multiple Devices (MD) Block Driver

The Linux RAID code uses a block device driver called MD (Multiple Devices) to group together collections of block devices into one logical device (`/dev/md[n]`). For RAID 0 this is very simple. It involves the following change to the `ll_rw_blk` (low level block read / write) system call. See the directory `drivers/md` in the Linux source code [13]. Rubini and Corbet give some explanation [11].
The `ll_rw_blk` routine conceptually looks like this:

```
ll_rw_blk (blocks){
    sanity-checks ();
    for-each block in blocks {
        make_request (block);
    }
}
```

It is modified to support striping (RAID-0) in this way:

```
ll_rw_blk (blocks){
    sanity-checks ();
    for-each block in blocks {
        if (block is-in md-device)
            md_map (block)
    }
    for-each block in blocks {
        make_request (block);
    }
}
```

### 2.7.2 QoS RAID as an extension to the MD driver

The `ll_rw_blk` call is augmented further to provide the QoS we need by rewriting the `md_map` function.

The new `md_map` takes a block number and returns a device id and the block number transformed by a QoS RAID sector remapping function `qmap(b)` of the block address. Obviously there will be a computational overhead proportional to the complexity of `qmap`. We will discuss this later.

The implementation is at the block address low level so any file system can run on top.

### 2.7.3 Geometry Issues

If the mapping function is going to be as accurate as possible we must know the physical geometry of the disk. As this information is completely hidden, we have to store a list of known hard disk geometries and compile the driver with disk specific parameters.

There are a couple of ways that `qmap` can get the logical geometry information. Storing the geometry at the time of compilation of the MD driver is fast to mount, easy to implement with slim code but inflexible. For example it would be impossible to use two QoS arrays of different disks. Better is to modify the user level utility `mkraid` to write a RAID superblock containing the geometry and another user space utility, `raidstart`, at boot time to read the superblock.

Worthington [18] shows that it is actually possible to extract the geometry of SCSI hard disks with surprising accuracy through a combination of *interrogation* (SCSI mode pages) and *empirical extraction* (timings). However this would restrict us to working with SCSI disks only. It also requires a high degree of timing accuracy which is not easy on most systems. Though this approach would be useful, it is unnecessary to test these algorithms, as I know the geometries of the disks I'm working with.

We should also provide another generic mapping function for use when the physical geometry information is not available. This will smooth the curve (of transfer rate against distance from spindle) by pairing fast and slow sectors in a linear manner. Of course sometimes one disk will be reading or writing for longer than the other, but the minimum transfer rate should still be higher than with a single zoned disk or RAID 0 array.

Throughout this document I will use `qmap_geo` to refer to the algorithm with knowledge of the physical geometry and `qmap_nogeo` for the "best-guess" algorithm.

### 2.7.4 Assumptions

There are a few fundamental assumptions I will make for the discussion of possible `qmap` algorithms.

**Ample bus bandwidth.** The PCI/Sbus and SCSI/IDE buses are not limiting the maximum transfer of both disks. On the test platforms that I am using (section 2.5) the PCI and Sbus both have a capacity large enough for two disks transferring at their maximum rates. The Sun has only the two disks in the QoS array on the SCSI controller and the PC has a UDMA/100 channel for each IDE drive.

**No bad sectors or relocations.** Section 2.6.4 discussed defect management. We cannot do much about this but with new disks we can assume there are few enough relocations such that the OS read buffering or the buffer in the streaming video codecs can accommodate the short drop in transfer rate due to the added seek time. Worthington [17] examined six disks and found defects caused "no noticeable effect" on disk performance.

**No computational overhead due to calculating mapping.** Clearly the `qmap` function will induce a computational overhead but it is reasonable to ignore it as the disk speed is slow compared to the CPU. After all, it is very small compared to the overhead of a journaling filesystem (such as ReiserFS or NTFS) on top.

### 2.7.5 Reading backwards, Biasing and Pre-fetching

The firmware in the disk is unaware that we are using it in a QoS array, so it will try to pre-fetch based on normal disk access patterns. As one of our disks is effectively reading backwards (in a saw tooth fashion) we will loose some of the effect of the pre-fetching; particularly if the disk cache is large. The saw tooth motion will also incur larger seek times on that device. This problem could be balanced by "biasing" the backwards reading drive. That is make its stripe sizes a little smaller than the forwards device.

### 2.7.6 `qmap_nogeo`

The naive algorithm simply writes one block to the *edge* of disk A and the next to the *spindle* of disk B and moves towards to *midpoint* of both disks.

This is throwing away any advantage gained by the device prefetching on disk B because it has to read backwards. Disk A reads ahead but as we are limited by the slowest block read time, this does not raise the lower bound on transfer rate at all. We have slashed the standard deviation of the access time at the cost of doubling the mean. We can do better.

To solve this problem we write a different number of blocks to each drive at a time. I will refer to this number of blocks as the **chunk size** or the **stripe size** interchangeably. If the chunk size written to drive A is $x$ then the number of blocks written to B is $f(a).x$ where $f(a)$ is a function of the block address, $a$.

This allows the "backwards" device to actually read forwards in a *saw-tooth* manner.

In RAID 0 the stripe size is constant to a particular array across all disks and all zones (zoning is irrelevant). A QoS stripe size is a function of the address of each device. This function is either linear or grows in discrete steps. **Logical zoning** (developed next) is an example of a discrete stripe size function.

A RAID 0 stripe is typically 16k (32 blocks). We will probably make our average stripe size larger (64 - 128k) to help the "backwards" device to pre-fetch. Constant transfer rates will still be delivered, as long as the requests sizes are large in comparison (e.g. $>$ 1mb).

### 2.7.7 Logical Zoning

We know that zones exist on the disk even if we can't easily deduce the boundaries. So why not impose our own logical geometry with zones of sizes to suit our requirements?

By pairing the inside zone of one disk (lets call it *disk A*) with the outside zone of another disk (*disk B*), the next zone out on *disk A* with the next one in on *disk B* and so on, we can create a stripe, putting more blocks in the fast zone than the slow zone and hence averaging our transfer rate across the logical disk. All we have to do is make sure our "logical" zones are the correct sizes.

The logical zones will not exactly match up to the actual boundaries but the closer the number of logical zones is to the number of physical zones, the better the boundary match obtained.

Let $z$ be the number of zones we divide the logical disk into where $z \mod 2 = 0$ (ie $z$ is an even number). $p_{max}$ and $p_{min}$ are the number of blocks per track in outside and inside zones of the physical geometry, respectively. Zone $i = 0$ is the innermost zone. $T$ is the total number of blocks on the physical disk.

If we write one "chunk" of blocks to the inside zone, we will write some multiple of that size ($b_0$) to the outside zone. Of course for each zone pairing, this multiple will be different. In fact it will get smaller by a linear function until it is 1 for the midmost pairing (we write equal chunk sizes in the middle zones). Equation 2.1 gives the increment, $dr$ by which the multiple increases from 1 as the zone pairings separate.

$$dr = \frac{(\frac{p_{max}}{p_{min}}) - 1}{(z/2) - 1} \tag{2.1}$$

Now we are in a position to define this multiple, $r$ in terms of the smallest member of the zone pairing, $i$.

$$\text{ideal multiple, } r^{ideal}(i) = \frac{p_{max}}{p_{min}} - (i.dr) \tag{2.2}$$

However, the stripe width needs to be a whole number of blocks so we adjust equation 2.2 by shrinking each zone to a multiple of the minimum stripe unit, or **blocking factor**, $blocking$. In equation 2.3 $\lfloor ... \rfloor$ means $floor$.

$$\text{block exact multiple, } r_i = \frac{\left\lfloor r^{ideal}(i).blocking \right\rfloor}{blocking} \tag{2.3}$$

Given this formula we can express the size of each of our new "logical" zones, $b_i$. The first case in equation 2.4 accounts for each of the smaller members of zone pairings. The second case, for the large member of the pair, can be calculated from the appropriate ratio and the size of the smaller pair member.

$$b_i = \begin{cases} \frac{2.T}{z} \cdot \frac{r_i}{r_i+1} & 0 < i < z/2 \\ b_{(z-1-i)} \;/\; r_{(z-1-i)} & z/2 <= i < z \end{cases} \tag{2.4}$$

Again, in an implementation we must ensure that each zone size is an exact multiple of $blocking \,.\, r_i$.

Figure 2.4 shows an example using values similar to the IBM disk in figure 2.3.

| **Zone Pairings** $i \Leftrightarrow z-1-i$ | $r^{ideal}$ | $r$ *(block exact)* | **Blocks in Zone** $i\ (z-1-i)$ | **Blocks in Zone** *(block exact)* |
|---|---|---|---|---|
| $0 \Leftrightarrow 13$ | 2.162 | 2.125 | 4884005 (2258852) | 4857138 (2285712) |
| $1 \Leftrightarrow 12$ | 1.968 | 1.875 | 4736614 (2406243) | 4658385 (2484472) |
| $2 \Leftrightarrow 11$ | 1.775 | 1.750 | 4568646 (2574212) | 4545450 (2597400) |
| $3 \Leftrightarrow 10$ | 1.581 | 1.500 | 4375467 (2767390) | 4285704 (2857136) |
| $4 \Leftrightarrow 9$ | 1.387 | 1.375 | 4150943 (2991914) | 4135329 (3007512) |
| $5 \Leftrightarrow 8$ | 1.194 | 1.125 | 3886770 (3256087) | 3781512 (3361344) |
| $6 \Leftrightarrow 7$ | 1.000 | 1.000 | 3571429 (3571429) | 3571424 (3571424) |

Figure 2.4: Logical zoning example

$z = 14;\ T = 50,000,000;\ blocking = 8;\ p_{max} = 800;\ p_{min} = 370$

The sum of the logical zone sizes is 49,999,942. We have had to lose 58 blocks due to truncating to a multiple of the blocking unit.

Given a logical block address, $x$, into the RAID array, we need a mapping algorithm to convert it into a device id (A or B) and a logical block address for that device. This is such an algorithm:

1. Deduce the logical zone pairing that $x$ is a member of, and hence the stripe (chunk) sizes $s_A$ and $s_B$.
2. Calculate the quotient, $q$ and the remainder $m$ of $\frac{x - start_block(zone_x)}{s_A + s_B}$.
3. If $m \geq s_A$ the device id is B, otherwise it is A.
4. If $m \geq s_A$ the offset into the stripe (chunk) is $m - s_A$, otherwise it is $m$.
5. $q\ .\ s_{dev}$ gives the start block of the stripe, where $dev$ is the device id.

The actual address is given by

$$start_block(zone) + start_block(stripe) + offset$$

It turns out that we don't actually need the two values $p_{max}$ and $p_{min}$ as it is only ever the ratio $\frac{p_{max}}{p_{min}}$ that we use. This is particularly easy to measure; we simply measure the time taken to write a few tracks of data to the outside and inside of the disk, recording the ratio. An average over about 30 trials gives a result to within manufacturers tolerances of rotation speed.

One advantage of logical zoning is the small amount of information that defines a particular array. All we need to store in the superblock is $z$, the ratio $\frac{p_{max}}{p_{min}}$ and the blocking factor as the size of the device ($T$) can be obtained from the `ioctl` call `BLKGETSIZE`. The other main advantage over a solution that uses a linear stripe size function is that (within a zone at least) there is no "backwards" reading drive so seeks are reduced and device prefetching can be fully exploited: or almost fully, depending on the request size and onboard cache capacity.

### 2.7.8 `qmap_geo`

Building on our development of logical zoning, we should be able to get a more even distribution of transfer rates if we use the *actual* physical zones. We need to change the multiple $r_i$.

$$\text{multiple, } r_i^{geo} = \frac{\#sectorsinouterzone}{\#sectorsininnerzone} \quad (2.5)$$

## 2.8 Media Streams

In order to record disk traces for a real streaming media server I needed to find a program that would stream multiple random access video feeds. A suitable product is "ffserver" [7] part of "ffmpeg". After learning how to use it, I set it up a script (Appendix D) to stream 50 DivX 4 videos at 700kbps. This results in 34mb/s which is more than my disks can sustain; leading to slow, non real time videos.

# Chapter 3

# Implementation

## 3.1 Configuring DiskSim

DiskSim is an executable program that reads a configuration file and generates an output file. The configuration file(s) contains a specification of the disk drives used (several are including with DiskSim), a specification of the architecture inter-connection, a list of require statistics to be output and an optional specification of the synthetic workload.

We will run the simulator on a single disk, a two-disk RAID 0 array, a QoS RAID array with no geometry knowledge and a QoS RAID array with geometry knowledge.

**I/O Subsystem Interconnection Specifications** DiskSim is configured with two identical IBM 18es SCSI drives on the same bus to a simple controller connected in turn to a device driver. Each physical drive is mapped directly to a logical partition. All QoS RAID and RAID 0 mappings in the test are performed directly on the traces by Perl scripts.

**Output Specifications** The statistics we are interested in are the distributions of the request completion time for each of the four configurations (single disk, standard stripe, and each QoS stripe). In particular we will be looking at the **maximum**, **mean** and **standard deviation**.

Other statistics of interest are the **disk buffer hit ratio** and the distribution of **disk seek times** of each array member for judging the effect of saw tooth reading (see section 2.7.5).

## 3.2 Recording of Simulation traces

I used my own low level SCSI trace logger (section 3.7) to trace the operation of `ffsever` as described in section 2.8. Appendix D shows the Bash script that was used to start all the streams and trace the disk. The program `leakyWall` was written to receive the bytes and wait for a certain period (based on the given bit rate) and loop. If bytes are not ready it displays a warning. This simulates a media client.

## 3.3 Application of mappings to simulation traces

The Logical Zoning algorithm was implemented in a Perl script that reads requests from STDIN and prints re-mapped requests on STDOUT. It is configured by command line

switches and can print extra information on STDERR. A validation script was used to check the integrity of the algorithm (section 4.2). These scripts are included with the rest of the source code. See section 3.4 for download instructions.

## 3.4 Download and Installation instructions

The implementation described in this chapter can be obtained from the following locations:

- From the U.K. - `http://www.thor.cam.ac.uk/~odl21/qosraid.tar.gz`
- From the U.S. - `http://owainlloyd.com/qosraid.tar.gz`

To unpack the archive type:

```
gunzip < qosraid.tar.gz | tar -xvf -
```

Change to the first new directory created, containing the user space RAID tools, run the configure script, compile and install with:

```
cd raidtools
./configure
make
make install
```

Now patch the kernel sources from the patch file in the second directory created from the tar ball. For example if you unpacked the archive into your home directory and have the Linux 2.4.17 kernel sources in `/usr/src/linux`.

```
cd /usr/src/linux
patch -p1 < ~/qosraid/kernel_patch-2.4.17
```

Compile the kernel with RAID support, boot it and create a QoSRAID array. To do this you just need to create a configuration file in `/etc/raidtab`, run `mkraid` and `raidstart`. See the file `README` in the `raidtools` source directory for instructions.

## 3.5 The Kernel Patch

The first problem came when I discovered floating point arithmetic can't easily be done in the kernel. This meant I had to ship the calculation of zone sizes out to the `raidtools` code and use an `IOCTL` call the pass the values over when starting the disk array.

The QoS personality was added in the file `/drivers/md/md.c` and support for the extra `ioctl` added. `md.c` implements the general RAID block driver.

The personality specific code for Logical Zoning was added to a new file `qos_raid.c`. This includes a function for status reporting to `/proc/mdstat`, functions for starting and and stopping the array and most importantly the block request function that performs the remapping. A version of this function with the sanity checks removed is shown in Appendix B.

Figure 3.1 shows the C source files that make up the core QoS RAID implementation and how the kernel interacts with user code. The boxes shaded in grey are executables.

I added extra code that checks for the mapping making sense and aborts the request, logging a bug to "syslog" if something seems dangerous.

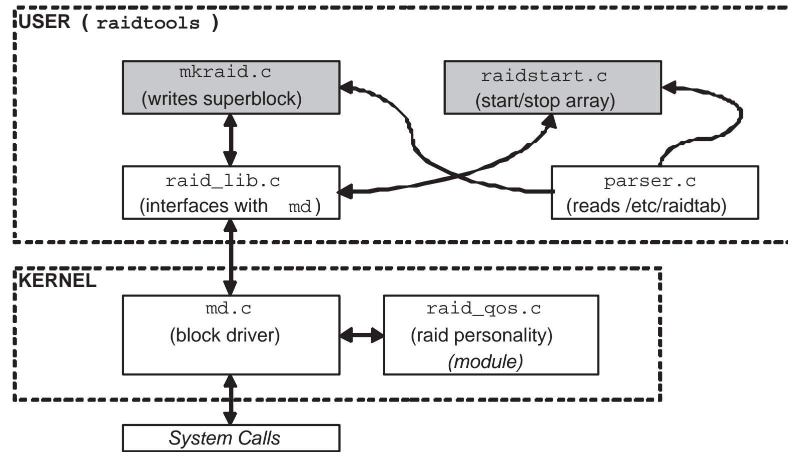


Figure 3.1: The layout of source files in the Linux implementation

It was very important that this code executed as fast as possible because every disk request must pass through it. The RAID 0 code is very simple and only needs to perform addition, shifts and two divisions [13]. To implement the remapping function in C (Appendix B), I tuned the algorithm used in the Perl script to use shifts and bitwise operations where possible. The result performs four integer divisions, one multiplication and one modulus operation. The memory arrays of zone starting addresses and zone sizes are also accessed but these are small enough to fit in cache.

If these arrays had been bigger and might have caused cache misses, I would have either performed all the calculations on each request and cached the previous two results (i.e. the size and start of the current working zone pair), or just used the start address array and calculated the sizes on demand. An alternative solution that would have the speed advantage of the current code but not suffer so much if the arrays were lost from the cache would be to organize the arrays in such a way that the first element referred to the outside zone and the second element to the inside zone and so on. Due to spatial and temporal locality of disk reference, zone pairs are looked up together. Therefore only part of the array must remain in cache. It is better to only access one part of the array than constantly access its extremes. Figure 3.2 illustrates this. The top diagram represents the standard way of organizing the array with the lower diagram storing likely accessed elements adjacent.

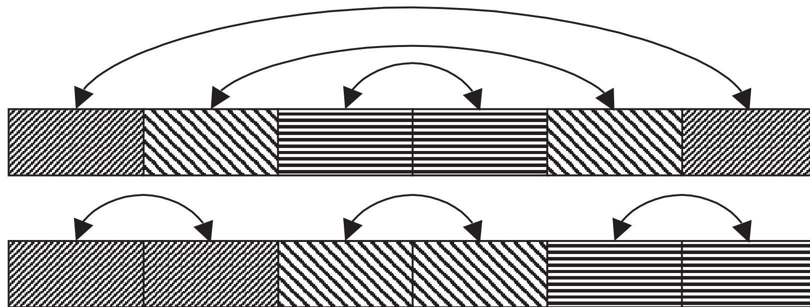

Figure 3.2: A cache efficient array layout

To allow the kernel to be configured with the new personality, the file `Config.in`

and the Makefile were altered. A patch file was created of the modified kernel against a vanilla source tree using the diff utility.

Debugging the kernel code was not easy. Changes to the personality could be tested by reloading the module (if the module could be unloaded). Some serious errors including kernel panics and any changes to the md driver needed a reboot. Using a VMWare virtual machine helped a little here. It also prevented permanent disk damage as initial testing was done on virtual disks.

## 3.6 The Raidtools Package

The user space utility mkraid is used to create a /dev/md[n] logical partition. It reads a configuration file (usually /etc/raidtab) and writes a superblock to the end of each constituent physical partition.

Modifications were made to the parser to accept the required new syntax in /etc/raidtab:

**raid-level qos -** qos is accepted in place of the usual 0, 1, 5, linear etc.

**geometry-knowledge -** 1 to use the qmap_geo algorithm, 0 to use qmap_nogeo.

**logical-zones -** The number of logical zones to divide to disk into - this must be an even number.

**max-blks and min-blks -** These are two integers used to express the quotient of outside / inside number of blks. A tool is supplied to calculate these.

**blocking-level -** This the a minimum stripe size that will be used in kb.

```
raiddev /dev/md0
        raid-level                    qos
        nr-raid-disks                 2
        persistent-superblock         1
        chunk-size                    8
        geometry-knowledge            0
        logical-zones                 24
        max-blks                      10
        min-blks                      6
        blocking-level                8

        device                   /dev/sda1
        raid-disk                0
        device                   /dev/sda2
        raid-disk                1
```

Figure 3.3: An sample /etc/raidtab for Logical Zoning

**Detection of device parameters:** A program that times many reads from the outside and inside of a block device (getQosParams) is used to measure the critical ratio for the configuration file entries max-blks and min-blks.

**Superblock write and read code:** The RAID superblock specification was altered using reserved space to store six more integers representing: the algorithm, the device type, the number of logical zones, the max over min quotient and the blocking level.

A suitable command to make a new array and start it would be:

```
mkraid --really-force /dev/md0
```

**raidstart:** is the program used to start a disk array. `raidstop` is a symlink to the same executable and is used to stop the array. The program uses ioctl calls to communicate with the md driver. My version of `raidstart` is modified to calculate the array of logical zone start addresses and pass it to the md driver before starting the disk array. The modified md driver expects this. This function is shown in Appendix C and can be compared with the original development of the Logical Zoning algorithm in section 2.7.7.

## 3.7 The Low Level Disk Trace Logger

In Chapter 2 it was shown that I must implement my own low level disk trace logger as no suitable product existed.

The simplest way was to:

- Modify the kernel boot code to reserve a large contiguous memory buffer.
- Modify the SCSI code to log to this buffer.
- Write a char device driver kernel module to read from the buffer.
- Write a user space utility to read from the driver and to start / stop logging and configure the device to be logged. (you don't want to log the device where the log is being written!)

Appendix A shows the basic code for the kernel modifications. The `alloc_bootmem` call is used in `init/main.c` to reserve the buffer. Global pointers to the read and write position and a wait queue are also set up here. It is crude because the memory is reserved at boot time and cannot be freed; but there is no other way to reserve a large contiguous block.

The SCSI block request function `scsi_request_fn` in `drivers/scsi/scsi_lib.c` contains extra code to fill this buffer with a "struct" of timestamp, block number, request size and request type (read or write) for each request sent to the device being logged. The resolution of the timestamps (using `do_gettimeofday`) on an x86 machine is microsecond [10].

The module creates an entry in `/proc` using the `create_proc_read_entry` call to display the buffer size, its maximum fill level, the device being logged (if any) and other useful debug info.

A node is created in `/dev/ll_trace` that supports the `read` system call to output a stream of the log entries, sleeping on the wait queue.

`IOCTL` system call commands are supported to start and stop logging, and set the logging device.

A user space utility (`lltrace`) is used to control the driver and convert binary traces into an ASCII file.

The `ll_trace` kernel patch, module and user tool can be downloaded from `http://owainlloyd.com/lltrace/`.

# Chapter 4

# Evaluation

In this chapter I explain how I have produced a working and usable product that gives better performance than existing solutions for high bandwidth non live streaming media servers.

**Packaged professionally:** The Linux implementation consists of a kernel patch and source tar ball of the modified user space utilities with a "configure" script and a "Makefile". This is the standard way to distribute code of this nature meaning any competent administrator should be able to try the code. Figure 4.1 shows the selection of the QoS raid module in a patched kernel using the "menuconfig" configuration interface.

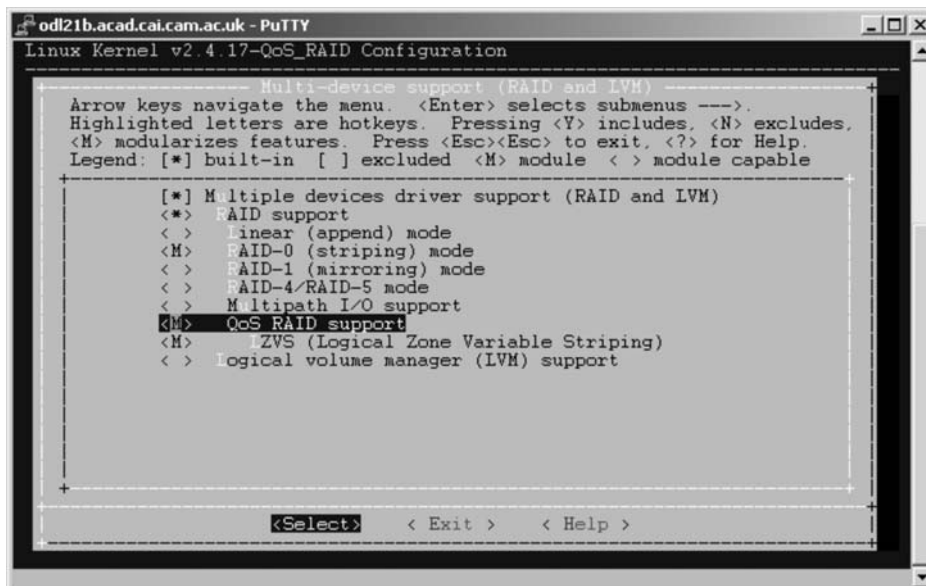


Figure 4.1: Configuring a QoS RAID kernel

The creation of a new LZVS (Logical Zone Variable Striping) array would be made easier if the `mkraid` tool performed the measurement of outside blocks to inside blocks itself. This is currently done using the supplied `getQosParms` program. Further benefit would be gained if the number of logical zones could be measured automatically by `mkraid`. This could be done by accurately timing reads across the disk and seeing how

many discrete jumps there are [18]. This would remove three of the required parameters in the configuration file `/etc/raidtab`.

**Fault free:** I have been running a QoS RAID array since February 2002 for storing movies and MP3 audio files. My version of the kernel is configured to log any suspect or failed requests. I also ran an intensive 24 hour test loading the array to its limit with writes and reads. This resulted in no problems. While I am not trying to claim there are no bugs at all, I believe this code is usable in a production environment as I have not yet had any problems.

**Failed to implement zone knowledge:** I failed to implement the version of my algorithm that uses the knowledge of the physical geometry. The main reason for this is that the pairs of drives I have are not actually identical. Although they are the same model number, measurement showed (and a careful look at the IBM documentation confirmed) that the zone boundaries were different. The simulations showed this mapping function would have given a small increase in performance, however the general case is of more practical use.

**Trace Driver:** The implementation of a working SCSI Trace driver was not an original consideration. However it is a useful product that many people across the world have downloaded from my web site since February. I have discussed it with a few of these people, none of whom have reported any problems with it. It is similarly packaged with a kernel patch and Makefile.

## 4.1 Results of Simulation

The DiskSim results do not allow us to see the exact time for a request to complete given its block address. They do however summarize the request completion times (response times) as show in figure 4.2. The traces used were recorded during the serving of streamed media using the method described in section 3.2. Each trace consisted of 15 thousand requests and was processed to represent a single disk, a RAID 0 array, a LZVS array and a QoS array with correct zone boundaries (`qmap_geo`). The trace represented considerable load on the disks (a mean of 31 outstanding requests on the single disk) which accounts for the large maximum response time of the single disk.

**Summary of Simulation of Request Completion times**
Trace size: 15,000 requests
Units: milliseconds

| | Single | RAID 0 | LZVS | qmap_geo |
|---|---|---|---|---|
| **std dev** | 92.1 | 71.2 | **14.8** | **12.8** |
| **mean** | 663 | 397 | **426** | **422** |
| **max** | 1192 | 619 | **520** | **504** |

| Percentage Performance Bound gain: (LVZS against RAID 0) | 16% |
|---|---|

Figure 4.2: Summary of Simulation Response Times

**Standard Deviation:** The RAID 0 array performs significantly more consistently than the single disk (23% lower S.D.) indicating that the trace was simply too demanding for

the single disk. The value of interest is the huge difference between the QoS arrays and the RAID 0 array. This shows that the data is more evenly distributed over the performance of the component disks. Exploiting the geometry knowledge makes gives a further improvement in this load balancing measure of 13%.

**Mean:** We see the LZVS and the geometry knowledge enhanced version are 7.5% and 6.5% worse, respectively, on average performance. This is what would be important for a more traditional storage use such as a file server or root file system.

**Maximum:** The all important upper bound. The LZVS array performs 16% better than the RAID 0 array, with the geometry knowledge version achieving a 18.5% improvement.

A graph representing the summary is shown in figure 4.3.

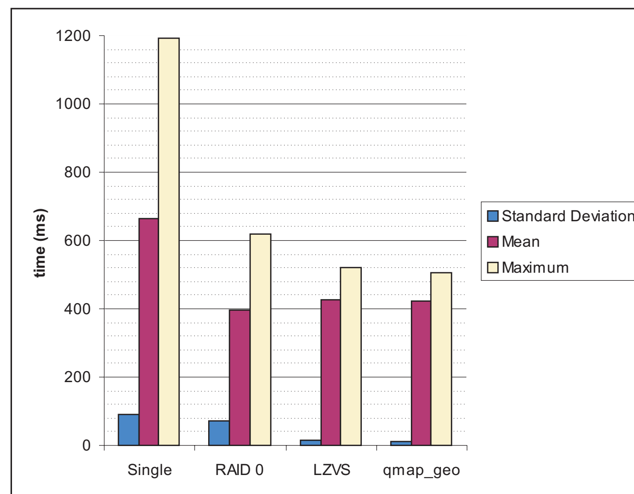


Figure 4.3: Graph to show summary of Simulation response times

**Seek Times:** Figure 4.4 shows the mean, standard deviation and maximum seek times from a simulation of 15,000 disk requests. It shows the values for a standard RAID 0 array and a LZVS array. Also it shows the individual components of the forwards and backwards reading devices for the LZVS array. The higher average and maximum values

**Seek Times (ms):**

| | | LZVS | | |
|---|---|---|---|---|
| | **RAID 0** | **Array** | **Forwards** | **Backwards** |
| *Average* | 6.7 | 7.0 | 6.6 | 7.4 |
| *Std. Dev.* | 3.3 | 3.5 | 3.3 | 3.6 |
| *Maximum* | 12.7 | 13.1 | 11.3 | 14.8 |

Figure 4.4: Simulation Seek Times

for the backwards drive against the forwards drive confirm the suggestions made in section 2.7.5. The result is that the maximum seek time for a LZVS array is higher than for a RAID 0 array. As the seek time is a component of the request completion time, this is an

extra overhead for LZVS. The overhead is constant and hence will have less effect on a continuous transfer of 8mb than one of 8kb.

## 4.2 Validation of the Mapping Functions

I needed to check the integrity of the algorithms, essentially whether they provide a 1-1 mapping. It is a serious problem if two logical addresses map to the same physical block. Of course, this has nothing to do with the semantics of the mapping — that is tested later in this section.

A Perl script was written that uses two associative arrays to represent the disks. An ordered list of single block requests for every logical block is generated. The list is fed through the LZVS (Logical Zone Variable Striping) Perl script and the logical address is stored in the mapped location. The re-mapped trace is then processed again, this time checking the arrays for a consecutive block number.

A similar C program was written to test the kernel code using the real disks instead of memory arrays.

Both the kernel implementation and the trace remapping script passed the test.

## 4.3 Measurement of Implementation Statistics

A C program was written to time read requests to a given device starting at a given block. The size of each read was 8mb and the measurement was performed five times recording the average. The maximum deviation among the five tests was less than 3%. The buffer cache was invalidated by reloading the SCSI module after each measurement. The test machine for these particular results was the Sun with dual IBM SCSI disks.

## 4.4 Implementation Results

**Distribution of request completion times:** Figure 4.5 shows the distribution of request completion times across the surface of the disk.

| | Block Address (Normalized to range 0-100) | Request Completion Time (ms) | | |
|---|---|---|---|---|
| | | Single Disk | RAID 0 | LZVS |
| Outside | 0 | 433 | 259 | 360 |
| ↑ | 5 | 438 | 261 | 361 |
| | 10 | 455 | 268 | 362 |
| | 15 | 468 | 273 | 361 |
| | 20 | 486 | 287 | 360 |
| | 25 | 489 | 286 | 346 |
| | 30 | 508 | 305 | 356 |
| | 35 | 516 | 310 | 350 |
| | 40 | 526 | 329 | 351 |
| | 45 | 548 | 331 | 340 |
| | 50 | 552 | 342 | 340 |
| | 55 | 569 | 350 | 346 |
| | 60 | 578 | 367 | 350 |
| | 65 | 598 | 370 | 351 |
| | 70 | 602 | 379 | 361 |
| | 75 | 619 | 387 | 361 |
| | 80 | 640 | 399 | 363 |
| | 85 | 642 | 406 | 362 |
| | 90 | 660 | 421 | 361 |
| Inside | 95 | 668 | 429 | 362 |

| | Single | RAID 0 | LZVS |
|---|---|---|---|
| **std dev =** | **75.0** | **55.3** | **7.7** |
| **mean =** | **550** | **338** | **355** |
| **max =** | **668** | **429** | **363** |

Figure 4.5: Table of Request Completion Times

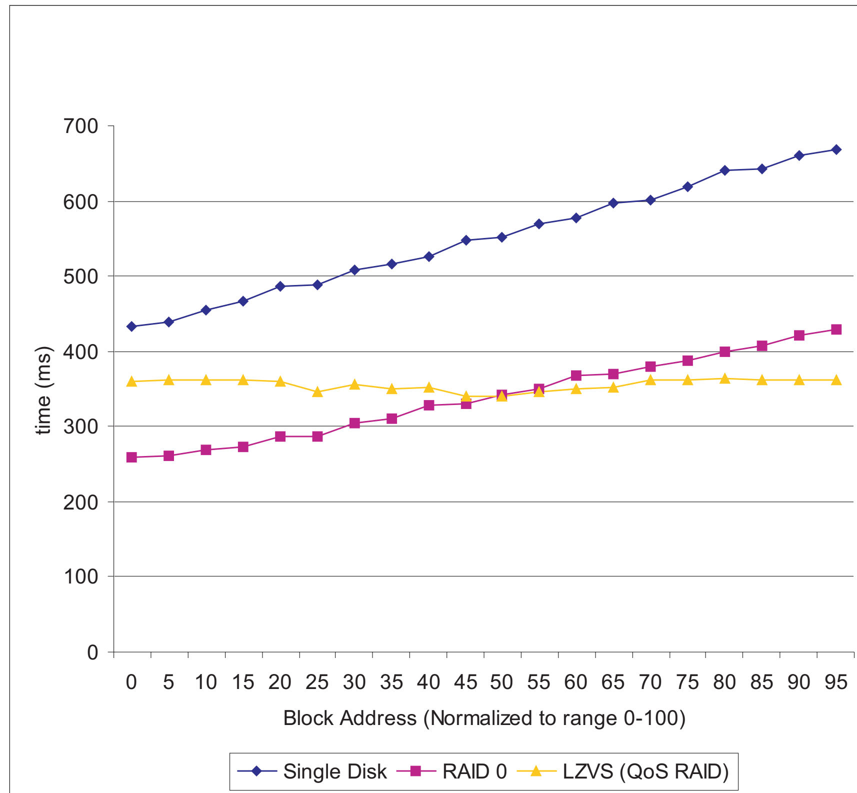


Figure 4.6: Graph of Request Completion Time against location of data on disk

**Mean, Maximum and Standard Deviation of request completion time:** As can be seen from Figure 4.5, the maximum time for a read request to a LZVS array at any point across the disk is 363ms against 429ms for a standard RAID 0 array of the same hardware. This is a 15% performance increase. That is, 15% more media streams can be served from the same hardware when guaranteeing a given bit rate.

The lower standard deviation (7.7ms against 55.3ms) translates to a more even, less wasteful distribution of performance. This can be considered as a measure of the load balancing [2].

The mean read time is 5% slower (355ms against 338ms). This is to be expected given the extra overhead required to calculate the physical device and address required by LZVS and the extra seek times. One would not want to use this type of array for a root file system.

### 4.4.1 Testing a streaming video server:

In section 2.3 on Evaluation Methodology I suggested a practical measure of the performance was to test a real streaming media server.

`ffserver` was used again to count how many random access streams could be served before a frame had to be dropped.

I use two machines for this test - one to serve from the array and one running the `leakWall` program described in section 3.2 to warn as soon as a data was not waiting for it. The files being served you distributed evenly across the entire disk array. The two

machines were connected by a 100mb/s network which was monitored to ensure it was not a bottleneck. Fortunately the aging drives on my Sun were no match for the network.

From a RAID 0 array 36 streams at 200kbps were achieved before `leakyWall` complained of missing data. The same configuration with a LZVS array managed 41 streams. This is a 14% percent improvement.

This result is in line with the predictions of performance made in the previous section by analyzing the speed of accessing the array at any address.

# Chapter 5

# Conclusions

I have developed the concept of Logical Zone Variable Striping (LZVS) (section 2.7.7) and demonstrated that it can be used to increase the load on a random access streaming media server given the same hardware, against a traditional disk array. Specifically a performance increase of 15% is expected (14% has been demonstrated in an applicable environment) over a two disk RAID 0 stripped array. Multiple LZVS arrays can be combined in RAID 0 arrays to compete with larger arrays.

The implementation meets the requirements outlined in section 2.1.

With the benefit of hindsight I would have concentrated on LZVS without thinking for so long that there was such a distinction between LZVS and its variant with exact geometry knowledge.

One particular area for improvement needed before the code could be considered stable is more extensive error checking. In particular, the values in `/etc/raidtab` are not checked (e.g. no warning is given if the number of logical zones is odd). These simple checks would make the code more robust to unfamiliar users.

With more time I would have liked to compare results from the Sun and the PC. The effect of slower processors, larger disk caches and smaller rotational latency could be studied.

In section 3.5 I mentioned the idea of calculating values as needed rather than risking cache misses when accessing arrays. It would be interesting to compare the performance of the current code with a version that does this.

In chapter 4 I suggested the implementation would be easier to use and more reliable if the `mkraid` utility performed some extra measurements. Along with an implementation of the algorithm with geometry knowledge, this would be an improvement.

## 5.1 Further Directions

Possible steps forward in this area of research that would have been too big for this particular project are listed here.

### 5.1.1 Specifying a variable QoS bound

Logical Zone Variable Striping (LZVS) allows you to guarantee one particular bound on the performance of the disk array. I imagine a development of this to be able to specify a set of partitions each with a given size and upper bound on transfer rate. This might be useful for audit trail or database logs (i.e. writes). An even faster part of the array could then be used for media streams or financial data feeds.

### 5.1.2 Extracting the PBN/LBN geometry mapping

A variant of LZVS with knowledge of the exact Physical / Logical mapping (i.e. zone boundaries) as discussed in this dissertation would be of far more use if accompanied by a tool to automatically determine this information. Bruce Worthington [18] describes methods for doing this but none are simple and all require a very accurate timer resolution.

### 5.1.3 Hardware that reads backwards

If the drive manufacturers could make a version of their devices that was addressed from the inside to the outside as well as traditional version, the performance could be improved by further exploiting pre-fetching and internal scheduling algorithms.

# Bibliography


[1] *Parallel Data Lab (Carnegie Mellon University)*. http://www.pdl.cmu.edu/.

[2] G. Ganger, Worthington B., Hou R., and Patt Y. *Disk Subsystem Load Balancing: Disk Striping vs. Conventional Data Placement*. Proceedings of the Hawaii International Conference on System Sciences, 1993.

[3] G.R. Ganger. *Generating Representative Synthetic Workloads: An Unsolved Problem*. Proceedings of the Computer Measurment Group Conference, 1995.

[4] G.R. Ganger, B.L. Worthington, and Y.N. Patt. *The DiskSim Simulation Environment Version 2.0 Reference Manual*. University of Michigan, 1999.

[5] *HP Labs Storage Systems Program*. http://www.hpl.hp.com/SSP/.

[6] IBM. *Deskstar 40GV and 75GXP Product Manual*. International Business Machines Corporation, 2000.

[7] G. Lantau. *FFmpeg Streaming Multimedia System - Multimedia Encoder and Streaming Server*. http://ffmpeg.sourceforge.net, 2001.

[8] *The linux-raid mailing list*. To subcribe: majordomo@vger.kernel.org. http://groups.google.com/groups?group=mlist.linux.raid.

[9] C. Lu, Alvarez G., and Wilkes J. *Aqueduct: online data migration with performance guarantees*. USENIX Conference on File and Storage Technology (FAST'02), 2002.

[10] V. Malik and Yaghmour K. *Time keeping in the Linux Kernel*. http://www.embeddedlinuxworks.com/articles/kernel_time.html, 2001.

[11] A. Rubini and J. Corbet. *Linux Device Drivers (2nd Edition)*. O'Reilly, 2001.

[12] C. Ruemmler and J. Wilkes. *UNIX disk access patterns*. USENIX Technical Conference Proceedings, Winter, 1993.

[13] L. Torvalds et al. *The Linux Kernel Sources 2.4*. ftp.kernel.org, 2001.

[14] J. Wilkes. *The Pantheon Storage-System Simulator*. Hewlett-Packard Laboratories, Palo Alto, 1996.

[15] J. Wilkes. *Traveling to Rome: QoS specifications for automated storage system management*. Proceeding of the International Workshop on Quality of Service, 2001.

[16] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. *The HP AutoRAID hierarchcial storage system*. ACM Transactions on Computer Science, 14(1), February, 1996.

[17] B.L. Worthington, G.R. Ganger, and Y.N. Patt. *Scheduling for Modern Disk Drives and Non-Random Workloads*. University of Michigan, 1995.

[18] B.L. Worthington, G.R. Ganger, Y.N. Patt, and J. Wilkes. *On-line extraction of SCSI Disk Drive Parameters*. University of Michigan, 1996.

[19] K. Yaghmour and M.R. Dagenais. *Measuring and Characterizing System Behavior Using Kernel-Level Event Logging*. Proceedings of the USENIX Annual Technical Conference, 2000.

# Appendix A

# Low Level SCSI Trace Driver

A kernel patch was written that logs disk requests to a buffer. The following code in `init/main.c` reserves a large contiguous memory buffer when the kernel boots.

```
#ifdef CONFIG_LL_TRACE
// reserve the memory
ll_trace_buffer =(ll_trace_req_t*)
    alloc_bootmem(sizeof(ll_trace_req_t)* LL_TRACE_BUFFER_SIZE);
// initialize some counters
...
// initialize a wait queue
init_waitqueue_head(&ll_trace_wait_queue);
// print the start address of the buffer to the console
printk("ll_trace buffer allocated at address %p\n",
    ll_trace_buffer);
#endif
```

The actual logging is done in the block r/w `scsi_request_fn` function in `drivers/scsi/scsi_lib.c`.

```
// if this is the device being traced
if (req->rq_dev == ll_trace_dev) {
    ll_trace_buffer[ll_trace_woffset].id = ll_trace_wid;
    do_gettimeofday(&ll_trace_buffer[ll_trace_woffset].time);
    ll_trace_buffer[ll_trace_woffset].startblock = req->sector;
    ll_trace_buffer[ll_trace_woffset].size =
        req->current_nr_sectors;
    ll_trace_buffer[ll_trace_woffset].flags =
        (req->cmd == WRITE) + ? 1:0;
    ll_trace_wid++;
    ll_trace_woffset = (ll_trace_woffset == LL_TRACE_BUFFER_SIZE-1)
        ? 0 : ll_trace_woffset + 1;
    if (ll_trace_woffset == 0)
        ll_trace_buffer_repetition ++;
    // wake up sleeping driver (reader)
    wake_up_interruptible_sync(&ll_trace_wait_queue);
}
```

A kernel module that implements a char device reads from the buffer. It creates an file `/proc/ll_trace` to display statistics.

A user space program is used to start, stop and print logging and to control which device is logged. It communicates with the char module via `ioctl` calls and the normal `read` call.

# Appendix B

# QoS RAID kernel code

The Logical Zoning RAID remapping function is the core of this project. It is located in `drivers/md/raid_qos.c` and looks a little like this. I have removed the safety and sanity checks for simplicity.

```
static int raidqos_make_request
            (mddev_t *mddev, int rw, struct buffer_head * bh)
{
    unsigned int zone, blocking_bits, sA, sB;
        // sA and sB are the stripe sizes
    long q, m;  // quotient and remainder
    unsigned long block, addr, offset;
    raidqos_conf_t *conf = mddev_to_conf(mddev);

    block = bh->b_rsector >> 1; // 1k->512byte blocks
    blocking_bits = ffz(~conf->blocking);
    zone = (int) block /
             ((conf->be_total_size<<1)/mddev->sb->log_zones);
    sA = ( (conf->zsize[zone] * conf->blocking) /
            conf->zsize[mddev->sb->log_zones-zone-1]);
    sB = (conf->blocking);
    q = (int)(block-conf->zstartblock[zone]) / (sA+sB);
    m = (block-conf->zstartblock[zone]) % (sA+sB);
    if (m >= sA) {
        bh->b_rdev = conf->dev1->dev;
        offset = m-sA;
        addr = q << blocking_bits;
    } else {
        bh->b_rdev = conf->dev0->dev;
        offset = m;
        addr = q * sA;
    }
    bh->b_rsector =
        ((conf->zstartblock[zone]>>1) + addr + offset) << 1;
    return 1;
}
```

When the array is started `build_arrays` is used to calculate the start blocks of each logical zone. The zone sizes are calculated using floating point arithmetic in user mode. See Appendix C.

```
static int build_arrays (mddev_t *mddev)
{
    int i, j, z, blocking;
    unsigned long s;
    raidqos_conf_t *conf = mddev_to_conf(mddev);
    z = mddev->sb->log_zones;
    blocking = conf->blocking;

    conf->zstartblock = kmalloc(sizeof(long)*z, GFP_KERNEL);
    if (!conf->zstartblock)
        return 1;

    // zone start blocks
    for (i=0; i<z/2;i++) {
        s = 0;
        for (j=0; j<i; j++)
            s += conf->zsize[j] + conf->zsize[z-1-j];
        conf->zstartblock[i] = s;
    }

    // total logical size
    conf->be_total_size = conf->zstartblock[z/2-1] +
            conf->zsize[z/2] + conf->zsize[z/2-1];
    // print info
    printk("raidqos : number of zones: %d\n",
                z);
    printk("raidqos : blocking level:  %d\n",
                conf->blocking<<10);
    printk("raidqos : total blocks:    %ld\n",
                conf->total_size);
    printk("raidqos : usable blocks:   %ld\n",
                conf->be_total_size);
    return 0;
}
```

# Appendix C

# Raid Tools Code

The following code found in `raid_lib.c` of the "raidtools" package is used when `raidstart` is run to calculate the logical zone sizes. The array it produces is passed to the kernel's md driver via an `ioctl` system call.

```
unsigned long *build_lzvs_sizes(md_cfg_entry_t *cfg)
{
    int i,
        z=cfg->array.param.qos_info.log_zones,
        blocking, fd;
    double dr, pmm, *ri, *r_be;
    unsigned long size,
        total_size=0,
        *zsize = malloc(sizeof(long) * z);
    blocking = cfg->array.param.qos_info.blocking;
    pmm = (double)  cfg->array.param.qos_info.max_blks /
                    cfg->array.param.qos_info.min_blks;

    for (i=0; i<cfg->array.param.nr_disks; i++) {
        fd = open (cfg->device_name[i], O_RDONLY);
        if (!fd) {
            printf("Cannot open %s!\n", cfg->device_name[i]);
            return NULL;
        }
        ioctl (fd, BLKGETSIZE, &size);
        printf("size of %s is %ld\n", cfg->device_name[i], size);
        total_size += size>>1;
    }

    // increment
    dr = (double) (pmm - 1) / ((z / 2) - 1);

    ri = malloc(sizeof(double)*z);
    r_be = malloc(sizeof(double)*z);

    // ideal ratios
```

```
    for (i=0; i<z; i++)
        ri[i] = (i < z/2) ? pmm - (i * dr) : ri[z-1-i];
    // block exact ratios
    for (i=0; i<z; i++) {
        r_be[i] = (double)((long)(ri[i]*blocking)) / blocking;
    }
    // zone sizes
    for (i=0; i<z; i++) {
        zsize[i] = (i<z/2) ?
            (long)((2*total_size/z)*(r_be[i]/(r_be[i]+1)) /
                (blocking*floor(ri[i]*blocking)))
            * blocking * (long)(ri[i]*blocking)
        :   (long)(( zsize[z-1-i] / r_be[z-1-i]) / blocking)
            * blocking;
    }

    free (ri);
    free (r_be);
    return zsize;
}
```

# Appendix D

# Tracing the Media Server

This shell script was used to start many media streams and record a disk trace. It also shows how to use my SCSI tracer. `leakywall` is a C program that acts like a media client by waiting before taking the next byte from input and warns if bytes are not ready.

```
#!/bin/bash
TRACE_FILE=01
LOGDEV=sdc1
DIVX_STREAMS=50
DIVX_DELAY=10

#reload kernel module
rmmod -s ll_trace; insmod -s ll_trace

#start ffserver using /etc/ffserver.conf and get pid
ffserver &
FF_PID=`ps -x | grep ffserver | grep -v grep | cut -d" " -f2`

#trace low level disk requests
lltrace reset
lltrace device /dev/$LOGDEV
lltrace start $TRACE_FILE.trace > /dev/null &
LLTRACE_PID=`ps -x | grep lltrace | grep -v grep | cut -d" " -f2`

#start clients
divxi=1
while [ "$divxi" -le $DIVX_STREAMS ] do
    echo "starting DivX stream $divxi"
    wget http://localhost:8090/test$divxi.avi -O - -q | \
            leakywall 700 divx_$divxi &
    sleep $DIVX_DELAY
    let "divxi+=1"
done
echo "ffserver PID is $FF_PID"
echo "lltrace PID is $LLTRACE_PID"
```