Aalto University
School of Science
Degree Programme in Computer Science and Engineering

Paul Tötterman

Performance and Scalability of a Sensor Data Storage Framework

Master's Thesis
Helsinki, March 10, 2015

Supervisor: Associate Professor Keijo Heljanko
Advisor: Lasse Rasinen M.Sc. (Tech.)

Aalto University
School of Science
Degree Programme in Computer Science and Engineering

ABSTRACT OF
MASTER'S THESIS

Author: Paul Tötterman

Title:
Performance and Scalability of a Sensor Data Storage Framework

Date: March 10, 2015 **Pages:** 67

Major: Software Technology **Code:** T-110

Supervisor: Associate Professor Keijo Heljanko

Advisor: Lasse Rasinen M.Sc. (Tech.)

Modern artificial intelligence and machine learning applications build on analysis and training using large datasets. New research and development does not always start with existing big datasets, but accumulate data over time. The same storage solution does not necessarily cover the scale during the lifetime of the research, especially if scaling up from using common workgroup storage technologies.

The storage infrastructure at ZenRobotics has grown using standard workgroup technologies. The current approach is starting to show its limits, while the storage growth is predicted to continue and accelerate. Successful capacity planning and expansion requires a better understanding of the patterns of the use of storage and its growth.

We have examined the current storage architecture and stored data from different perspectives in order to gain a better understanding of the situation. By performing a number of experiments we determine key properties of the employed technologies. The combination of these factors allows us to make informed decisions about future storage solutions.

Current usage patterns are in many ways inefficient and changes are needed in order to be able to work with larger volumes of data. Some changes would allow to scale the current architecture a bit further, but in order to scale horizontally instead of just vertically, there is a need to start designing for scalability in the future system architecture.

Keywords: storage, big data, performance, scalability

Language: English

Aalto-yliopisto
Perustieteiden korkeakoulu
Tietotekniikan koulutusohjelma

DIPLOMITYÖN
TIIVISTELMÄ

Tekijä:	Paul Tötterman		
Työn nimi:	Suorituskyky ja skaalautuvuus sensoridatan tallennuksessa		
Päiväys:	10. maaliskuuta 2015	**Sivumäärä:**	67
Pääaine:	Ohjelmistotekniikka	**Koodi:**	T-110
Valvoja:	Professori Keijo Heljanko		
Ohjaaja:	Diplomi-insinööri Lasse Rasinen		

Modernit tekoälyn ja koneoppimisen sovellukset perustuvat suurten tietomäärien analyysin ja käyttöön opetusdatana. Suuren aineiston olemassaolo ei aina ole itsestäänselvää tutkimuksen tai tuotekehityksen alkaessa. Samat tallennusratkaisut eivät välttämättä pysty kattamaan skaalautumistarpeita tutkimuksen koko keston ajalta, varsinkaan jos lähtökohtana ovat laajassa käytössä olevat työryhmätallennusratkaisut.

ZenRoboticsilla käytössä oleva tallennusinfrastruktuuri on kasvanut yleisiä työryhmätallennusteknologioita käyttäen. Nykyisen lähestymistavan rajat alkavat tulla vastaan, kun taas tallennuskapasiteetin tarve näyttäisi kasvavan ja kasvun tahti kiihtyvän. Tallennuskapasiteetin laajentamisen suunnittelu ja laajennuksen toteuttaminen edellyttävät parempaa käyttötapojen ja kasvun ymmärrystä.

Tämä diplomityö tutkii nykyistä tallennusarkkitehtuuria ja tallennettua dataa eri näkökulmista nykytilanteen parempaan hahmottamiseen tähdäten. Suoritetuilla mittauksilla selvitimme käytössä olevien teknologioiden oleellisimmat ominaisuudet. Yhdessä näiden perusteella pystymme tekemään tietoisempia valintoja tulevia tallennusratkaisuja koskien.

Nykyiset käyttötavat ovat monin tavoin tehottomia. Suurempien tietomäärien käsittelemisen mahdollistamiseksi on tehtävä muutoksia. Työ esittelee muutosehdotuksia, joilla olisi mahdollista skaalata nykyistä tallennusarkkitehtuuria hieman suuremmalle kapasiteetille. Horisontaalisen skaalautumisen mahdollistamiseksi vertikaalisen sijaan on kuitenkin otettava skaalautuminen huomioon koko järjestelmän arkkitehtuurin suunnittelussa.

Asiasanat:	tallennus, big data, suorituskyky, skaalautuvuus
Kieli:	Englanti

Acknowledgements

I wish to thank everyone who helped and supported me through the decidedly long process of writing my thesis.

Foremost, I would like to express my gratitude to my supervisor, Keijo Heljanko, for the generous amount of time and effort devoted to guiding me.

I am indebted to my patient friend, Tatu Kairi, for proofreading and liberally annotating my numerous drafts.

Warm thanks to Lasse Rasinen, for offering to be my advisor and for his Zen attitude and sage advice.

Thanks to my employer, ZenRobotics, for making it possible to research this topic so concretely and publishing my findings. Special thanks to my colleagues, who have explained matters unfamiliar to me on several occasions.

Finally, I am truly grateful for the support and encouragement of my family.

Helsinki, March 10, 2015

Paul Tötterman

Abbreviations and Acronyms

ARC	ZFS Adaptive Replacement Cache
BTRFS	B-TRee File System
CDDL	Common Development and Distribution License
CIFS	Common Internet File System, also known as Server Message Block
COW	Copy-On-Write
CPU	Central Processing Unit
DMU	ZFS Data Management Unit
ECC	Error Checking and Correction
FIFO	First In, First Out, a common queuing discipline
GB	Gigabyte, 10^9 bytes
GiB	Gibibyte, 2^{30} bytes
GPL	General Public License
HDD	Hard-Disk Drive
IOPS	Input/Output Operations per Second
kB	Kilobyte, 10^3 bytes
KiB	Kibibyte, 2^{10} bytes
LVM	Logical Volume Manager
L2ARC	ZFS Level 2 Adaptive Replacement Cache
MB	Megabyte, 10^6 bytes
MiB	Mebibyte, 2^{20} bytes
MTU	Maximum Transmission Unit
NAS	Network Attached Storage, file system -level remote storage
NCQ	Native Command Queuing
NFS	Network File System
PB	Petabyte, 10^{15} bytes
PiB	Pebibyte, 2^{50} bytes
PNG	Portable Network Graphics
POSIX	Portable Operating System Interface

RADOS	(Ceph) Reliable Autonomic Distributed Object Store
RAID	Redundant Array of Independent Disks
RAM	Random Access Memory
RPC	Remote Procedure Call
SAN	Storage Area Network, block-level remote storage
SATA	Serial Advanced Technology Attachment
SCSI	Small Computer System Interface
SHA	Secure Hash Algorithm
SLOG	ZFS Synchronous Log device
SPA	ZFS Storage Pool Allocator
SSD	Solid State Drive
TB	Terabyte, 10^{12} bytes
TCQ	Tagged Command Queuing
TiB	Tebibyte, 2^{40} bytes
XFS	Extent-based file system by SGI and later Red Hat
ZFS	Zettabyte File System
ZIL	ZFS Intent Log
ZRR	ZenRobotics Recycler

Contents

Chapter 1

Introduction

> In my opinion this problem of making a large memory available at reasonably short notice is much more important than that of doing operations such as multiplication at high speed. Speed is necessary if the machine is to work fast enough for the machine to be commercially valuable, but a large storage capacity is necessary if it is to be capable of anything more than rather trivial operations. The storage capacity is therefore the more fundamental requirement.
>
> Alan M. Turing, 1947 Lecture to the London Mathematical Society

Businesses have adopted information technology during the past half-century so thoroughly that it plays a significant role in most companies, and in virtually all technology companies. As Alan Turing predicted, the central benefit from computers is not their ability to quickly perform complex calculations, but rather to store and transfer information — computation capabilities are largely only significant when there is data to process. Thanks to growing storage capacities and declining costs, the industry is storing ever-increasing amounts of data. This has led to the introduction of the term *big data*; datasets with sizes that make standard and commonly available tools, equipment and approaches to process them inadequate.

The ZenRobotics Recycler (ZRR) is a robotic waste recycling system. It makes use of sensor fusion, computer vision, machine learning and artificial intelligence in order to autonomously recognize objects from an unstructured waste stream and sort them into recyclable fractions. The system combines the inputs from a wide variety of sensors into a coherent real time analysis of

the waste stream from which it picks and sorts objects.

The various visible spectrum and near infrared cameras, 3D laser scanners and wide spectrum sensors operate at a high resolution and frame rate and constantly produce large data streams — the raw data from the sensors is several gigabits per second. While not all of the data is stored, some of it is stored and transferred to a central location for analysis.

1.1 Problem Statement

ZenRobotics currently stores a significant amount of data. Still, the data is only collected from a handful of ZRRs in operation. Each installation differs from the others, by variations in the waste stream, physical properties and sensors, and thus needs a separate set of data for machine learning. With an increase in the number of installations the amount of required storage also grows.

The data accumulating has long since outgrown the capacity for storing all of it. Recent data is therefore kept and old data removed. The removal of datasets that have been subject to human annotation in addition to automated processes would, however, make little sense. Since they are not removed, the amount of retained data per installation constantly grows.

Merely archiving the annotated data would serve very little use. Changes to the computer vision and artificial intelligence algorithms invalidate previous machine learning results. Because of this, all of the annotated datasets have to be reprocessed for updated machine learning outputs. In addition, other kinds of analyses are being developed, some of which are run periodically, some that change and have to be rerun for all existing data and some that are just run occasionally. These place performance requirements on the storage that are not satisfied by archival media such as tape.

So far, the infrastructure has grown mostly using standard off-the-shelf components acquired at a moderate cost. The next expansion step is likely going to be more expensive. From a capacity planning perspective a better understanding of the need for growth is needed and which factors influence it.

The goal of this Thesis is to understand the current storage architecture and stored data at a sufficiently fundamental level to make meaningful deductions about its performance characteristics and scalability. We perform experiments using relevant technologies and measure key properties. As a result, we present problems with the current approach, suggest solutions, and propose technologies to choose for scaling to meet future demand.

1.2 Structure of the Thesis

This Thesis is structured in five chapters, starting with this introduction. We present a high-level background and the problem statement. Relevant hardware and software technology is presented in the second chapter, starting from lower layers with each section building on top of the previous ones.

The third chapter introduces the specific environment in which ZenRobotics operates. Next we present the methods for evaluating selected key performance characteristics in chapter four. Finally, we present our conclusions.

Chapter 2

Background

In this chapter, we will introduce relevant technologies and concepts that are not specific to this case study, but have reached wide use in the past and currently. The technologies are introduced in order from lowest level to highest, as they build on top of each other.

2.1 Mass Storage

Valuable data that takes effort or time to produce needs to be stored on non-volatile media. One important factor for choosing suitable storage in a business setting is cost per capacity. Businesses are typically not only interested in purchase price, but total cost of ownership, including e.g. cost of electricity.

Performance is also a factor. Storage performance consists of two components: bandwidth and input/output operations per second (IOPS). Bandwidth is often the primary focus: how long will it take to store or retrieve a certain amount of data. Unintuitively, IOPS can in many cases dominate performance. This is evident when the storage is accessed concurrently or depending on the algorithms employed, if the data consists of a large number of small items of data. Patterson noted [35] that improving latency, and thus IOPS, often helps bandwidth, while many other bandwidth improvements come at the cost of latency.

Different data can have very varying access patterns. Researchers at Facebook realized [54] that for their binary large objects, accesses to a given object decreased significantly with time. They measured the IOPS per TB of objects grouped by age and moved older objects to cheaper, less performant storage. This is an excellent example of how one has to identify the relevant dimension before comparing options. Even now, with SSDs having been on

the consumer market for years, some reviews only measure bandwidth without considering IOPS performance at all.

2.2 Hard Disk Drives

For storing comparatively large amounts of data that needs to be randomly accessed, there are currently no economically superior alternatives to hard disk drives (HDDs). Although tape storage can be over an order of magnitude cheaper when factoring in power cost [38], tape is not suited for random access.

HDDs have a long history in computing. While they started out relatively simple, they have grown more complex and modern drives can be seen as I/O specialized computers. Some modern features that can lead to unpredictable HDD performance are remapped sectors, error recovery, command queuing, buffering and caching. Krevat et. al [30] showed that modern HDDs exhibit small differences in performance, even within a population of the exact same model and revision.

With increasing HDD densities, the constraints for error-free operation have also increased. Vibrations can affect the mechanical positioning of the head. Variations in electro-magnetic interactions between the head and the disk surface or platter can cause bits to be read differently than they were written, a phenomenon commonly known as *bitrot*. Cosmic rays can cause bits to flip in RAM buffers or in transmission, not to mention possible bugs in the increasingly complex software [4].

To mitigate physical problems, HDDs store error checking and correction information in addition to user data. A variety of codings have been used, including Hamming [19], Reed-Solomon [42] and low-density parity-check codes [16]. If the error checking code indicates an error in the read block, the drive can retry reading in case the error was transient, or report the error to the requestor. Consumer drives typically spend more time trying to re-read blocks than enterprise drives, as enterprise setups often employ RAID, covered in Section 2.3, which can be used to reconstruct missing data [15]. The retries cause deviations to the seek and read patterns, incurring unpredictable latencies.

If a block is determined to be a permanent cause of errors, the drive firmware can mark it bad and remap it, using reserved space to store the contents meant for the problematic block. This again can cause unpredictable internal I/O operations and thus latency to the other operations.

Operating systems have long tried to schedule disk operations in intelligent ways, but have less visibility into the actual effects than drive firmware. In

order to remedy this, command reordering protocols called Tagged Command Queuing (TCQ) and Native Command Queuing (NCQ) [14] have been introduced. Both allow the operating system to schedule several requests to drives and let the drive determine the optimal order for completing them.

Buffering and caching is also present on many levels, including drives themselves. Due to limited operating system visibility into drive state and firmware algorithms, the effect of buffers and cache internal to the drive is hard to predict. Caching is also of limited effect for writes, as many software systems that deliver consistency take great care to introduce write barriers and flush data from cache to non-volatile media as a guarantee for durability.

One recent addition in the pursuit of ever-higher data densities for HDDs is shingled magnetic recording (SMR) [2]. Traditionally drives have used a read/write head with a magnetic field that is narrow enough to only affect the desired track. In the case of SMR, the head no longer has to fit a single track when writing, as long as the field is sharply defined on two edges forming a corner. With SMR, tracks are written closer to each other than before and the head is allowed to interfere with data on upcoming tracks, as long as data on previous tracks is left intact. Practical use of drives with SMR presents new problems: data can no longer be written to random positions, but instead has to be appended. To gain some random write capability, platters can be divided in zones separated by unused tracks, allowing writing to start at the beginning of any desired zone. Selecting a suitable size for zones is a compromise between capacity and random write capability [1]. If writes do not align with zones, affected zones have to be read, modified in memory and written in full, severely impacting performance.

2.2.1 Performance Model

The performance of a HDD is made of two main components. Sustained transfer rate describes the rate at which a drive can keep transferring data between the disk and the I/O interconnect port on the drive, assuming the drive read head is in a suitable location and the drive can thus avoid repositioning the read head. Drives usually cache data located near short reads, improving performance of subsequent reads. Write performance can also be improved by caching, as long as the write request does not have a strict requirement of the data surviving a power-loss event. A higher rotational speed can improve the sustained transfer rate of a disk, as long as the density of data on the disk remains unaffected and the drive can keep up transferring the data at the speed the disk is rotating.

Drives with high sustained transfer rate perform well with linear reads or writes. However, if the drive needs to transfer data to or from different

parts of the disk, e.g., to serve several programs concurrently, it needs to position the read/write head on the right track for each block. This is called the seek time. Despite finding the correct track, a transfer cannot start until the correct sector is under the read/write head. On average, the disk has to rotate half a turn for this to occur. This delay is called rotational latency.

In order to better understand the basic performance characteristics of HDDs, we need a model of the performance. Essentially a drive is a spinning disc combined with a read/write head. A simple model of a hard drive would be one with a given sustained transfer rate R_{sus} and a given average seek time, including average rotational latency, t_{seek}. Requests arrive in a FIFO queue, each consisting of a position and length. The drive removes the first request from the queue, seeks to the correct position on disk and performs the transfer at the maximum transfer rate. If there are tasks for the drive to perform it should be either seeking or transferring. Assuming a randomly distributed workload of fixed size blocks, we should observe average seek times.

The average throughput R_{avg} achieved over a sequence of operations of length l_{block} is described in Equation 2.1.

$$R_{\text{avg}}(l_{\text{block}}) = l_{\text{block}} \times \frac{1}{t_{\text{seek}} + \frac{l_{\text{block}}}{R_{\text{sus}}}} \tag{2.1}$$

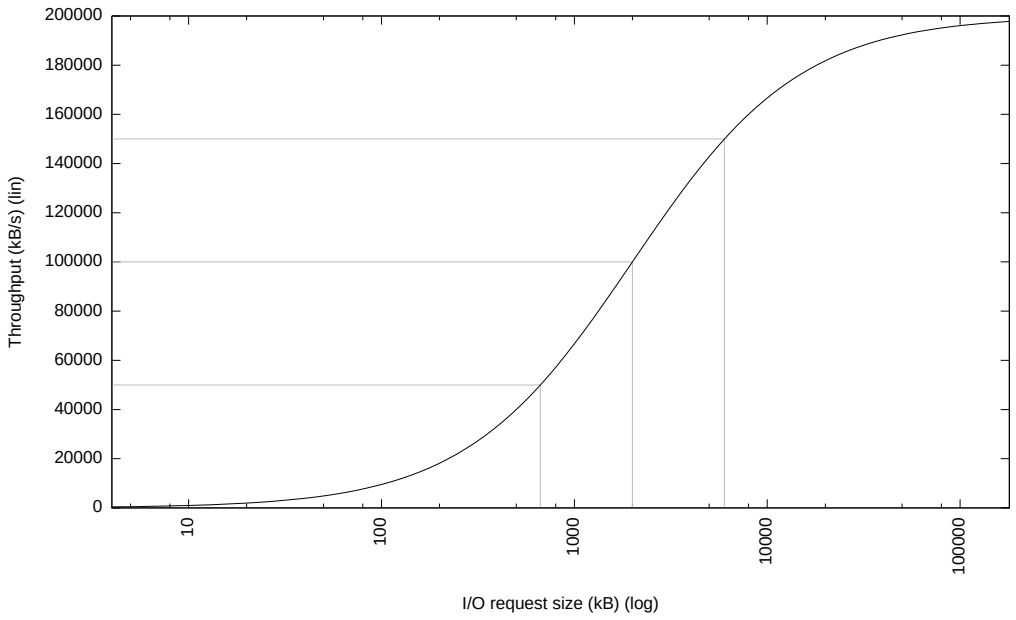

Figure 2.1: HDD throughput as a function of I/O operation size

Plotting Equation 2.1 with t_{seek} as 10 ms and R_{sus} as 200 MB/s we get Figure 2.1. It is clear from the figure that small request sizes have a severe impact on throughput. A request size of several megabytes has to be used in order to reach half of the sustained throughput. To reach 90% performance, the request size would have to be an order of magnitude larger.

2.3 Redundant Array of Independent Disks

Early hard drives were physically large, slow and expensive. Storage capacity of a computer system could be increased either by replacing a drive with a larger capacity drive, or adding a new drive and managing the location of data between the drives. Larger hard drives were not always available or economically feasible. At the same time hard drive performance did not increase at the same pace as that of other components, such as the CPU. Each additional component to a system increases the likelihood of a failure in the combined system.

Redundant Array of Independent Disks (RAID) was introduced by Patterson et al. [36] to combine the performance and capacity of several smaller drives. In order to achieve this goal without sacrificing uptime it also needed to solve the problem of larger probability of failure due to increased number of components. There exist both hardware and software implementations of RAID.

RAID combines several block devices, such as HDDs or SSDs, into one logical device or array. Typically the underlying devices are identical or at least have the same capacity but this is not strictly necessary; often the capacity of the smallest drive is used as a limit for all drives. Operations on the logical device are translated by the RAID layer into operations on the individual underlying devices. There are several different schemes for translating operations, called RAID levels. The most common currently used RAID levels are 0, 1, 10, 5 and 6.

RAID level 0 was not introduced in the original paper by Patterson et al. [36], but it plays a fundamental part in level 10. Level 0 or *striping* is not redundant at all. Instead, data blocks are spread evenly on all drives. A specific block of the logical drive is assigned to the drive block number modulo number of drives. Reads and writes spanning several drives benefit from the combined performance of the underlying drives. However, a failure in even one drive will likely break the file system of the logical drive.

RAID level 1 or *mirroring* is the simplest redundant RAID level. It provides the capacity of a single drive. The contents of the smallest underlying drive are copied to all underlying drives. If one fails, the data is still available

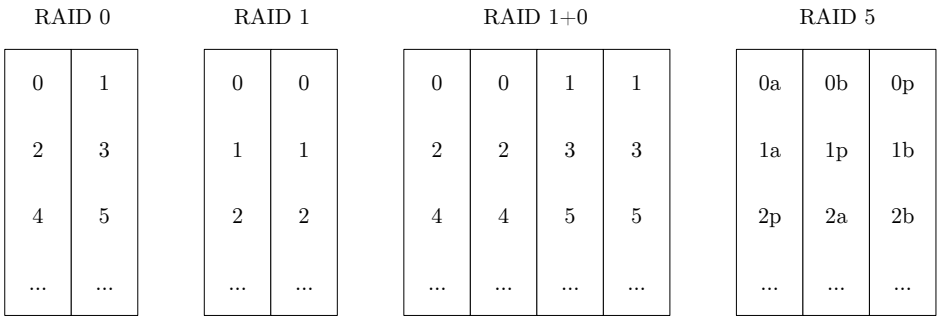

Figure 2.2: Different RAID levels

on the functioning drives. The faulty drive can be replaced and rebuilt by copying data from the functioning drives. Write performance is limited by the performance of the slowest underlying drive, as the same information has to be written to all drives, but read requests can be satisfied in parallel from all functioning drives.

RAID level 10, also known as 1+0, is a combination of RAID levels 1 and 0. RAID mirrors become the underlying devices of a larger striped array. This way the array can tolerate the malfunction of several drives as long as each mirror still has at least one functioning drive. The write performance is proportional to the sum of the write performance of all mirror sets. In the best case, read performance can be proportional to the sum of the read performance of all functioning drives in the array.

RAID level 5 is an improvement over no longer used RAID levels 2, 3 and 4. RAID levels 1 and 10 can tolerate the failure of drives, but the cost is at least 50% of the raw storage capacity. In RAID levels 2, 3, and 4 the drives are grouped into those that contain the data and those that contain error correction information that, in case of failure of a drive, can be combined with the remaining data drives to rebuild the missing data. RAID 2 does bit-level striping using Hamming codes, RAID 3 uses bit-level striping with XOR and RAID 4 uses block-level striping using XOR. They all require at least 3 drives, as does RAID 5. This improves usable capacity over RAID 1 and 10, but puts more strain on the drives containing error correction information, as those end up being rewritten whenever data on any data drive is modified. RAID 5 solves this by spreading the parity blocks evenly over all drives. The parity blocks of data stripes rotate, as shown in Figure 2.2. Read performance

on RAID 5 can reach the sum of the read performance of all drives, but even large sequential write performance is limited by the performance of the slowest drive.

RAID 6 is quite simply RAID 5 with two drives worth of error correction information and can thus tolerate the failure of two drives instead of one before data is lost. The two blocks of error correction information are calculated differently so that the data is still recoverable even if any two blocks are lost. Typically one is XOR like in RAID 5, because it is very simple to implement and performant while the other is more complex. Read and write performance remains similar to RAID 5. RAID 5 and 6 can both be combined with striping, like mirroring often is, and this is sometimes noted as RAID 50, 5+0, 60 or 6+0.

RAID 5 and 6 are widely used solutions when maximum performance is not required because of the low overhead redundancy. The design results in some complex behaviors that should be kept in mind called *write amplification* and *RAID write hole*.

A stripe in RAID 5 or 6 is the logical unit, consisting of blocks on all disks, for which the parity is calculated. If a write to the logical device covers a whole stripe, the new data can simply be calculated and written to all disks. However, if the write covers only part of a stripe, the old contents of the stripe have to be read, part of it replaced with new data, and the parity recalculated for the whole stripe. This procedure is called *read-modify-write*, and causes write amplification for small writes. Even a small one-sector write causes at least the sector itself and the parity to be written. The reads can be omitted, if the modified blocks are already present in cache.

Since modern HDDs are all unique [30], any notion of the equivalent command being performed identically on several drives must be discarded. RAID 2 and 3 assumes synchronous operation of drives in an array, but this has turned out to be impractical. Because of these individual differences, a stripe may not be atomically written by the drives working as part of an array. If a power failure were to occur during writing, the array has no information of which drives successfully committed the data to the disk. Because of this, the integrity of the stripe is left uncertain. This problem is usually referred to as the RAID write hole. A typical solution in hardware RAID implementations is to have a battery backed memory on the controller that retains the data until the power failure is resolved. This increases system complexity, as it cannot be implemented in software and batteries have to be maintained.

In addition to RAID, many environments use a logical volume manager (LVM) [21] to ease the administration of storage. Instead of partitioning a drive or RAID array into partitions, the volume manager is used to combine

drives (or partitions or logical drives) into volume groups from which space can be allocated to logical volumes. The LVM manages space to fulfill the administrator's requests to resize, transfer, or snapshot logical volumes.

2.4 Evolution of File Systems

During the earliest years of computing storage capacities were small and needs for storing various kinds of data were limited. The organization of data stored was uncomplicated — data was kept on a specific stack of punch cards or on a magnetic tape [57].

As capacities grew, the need to store more than one piece of data on a device surfaced and the first file systems were created. These first generation file systems can be characterized by the ability to name files contained on the same device in a single file system.

Storage capacities continued to grow and simple naming proved insufficient to organize the collection of files. Second generation file systems introduced hierarchical directories (often called folders in user interfaces) that solved the technical aspects of organizing files adequately for quite some time.

As the use of computing spread, computers themselves remained expensive — this also made shared use more common. With organizations such as the United States Department of Defense funding much of computing research, it did not take long for access control in the file system to become a desirable feature. Third generation file systems allowed for metadata such as file ownership and permissions. In POSIX-compatible systems, this metadata is stored in a data structure called an *inode*, which will be covered more closely in later chapters.

With the explosive growth of personal computing and thus increase of different failure modes during the final decades of the last millennium, file systems were often left in an inconsistent state by crashes. Fourth generation file systems improved the handling of such situations by introducing a journal, where intended changes are recorded prior to being performed. This way partial changes are much more reliably detected and properly handled after a crash.

Usually the performance penalty of journaling is not acceptable for all data and therefore only metadata is journaled. Because of this, files can still end up with partial writes in a crash. In addition, journaling does not protect against bitrot, described in Section 2.2.

One approach for solving these problems are *log-structured file systems* [44], where the whole file system is logically append-only. This has the desirable property of transforming random writes into sequential writes. In order to

efficiently use disk space, modified or removed file blocks are reclaimed in a process resembling garbage collection. Unfortunately, this resource intensive process is often triggered by low free space during writes, causing significant negative impact to performance.

Some recent file systems, like ZFS [8] and BTRFS [43], use copy-on-write and checksumming to solve these problems. This Thesis considers them to be fifth generation file systems.

2.5 Zettabyte File System

The Zettabyte File System (ZFS) [8] was made available by Sun Microsystems in 2005 following years of development that started in 2001. It has been called the only production quality fifth generation file system.

ZFS was designed with the benefit of hindsight of how contemporary file systems have scaled with the increase of storage capacity. Some limits from older file systems were thus completely avoided. Instead of having to specify the number of inodes at file system creation time, ZFS allocates inodes dynamically. Other limits, such as file and file system size, were raised so high that they are unlikely to be a problem for the foreseeable future.

Designing ZFS from a clean slate allowed not only for removing unnecessary limitations and taking into account advances in storage devices, but also for reconsidering the accumulated layers in the storage stack. While few of the ideas and features of ZFS were novel or unique, the combination of them, however, is. AdvFS [20] on Tru64 had storage pooling and NetApp Write Anywhere File Layout [25] had snapshotting, but not checksumming. LVM [21] provided storage pooling, but individual file systems still had to be resized. Log-structured file systems use copy-on-write, but garbage collection often makes them impractical.

ZFS has a layered architecture, but with different boundaries and interfaces between layers compared to the conventional combination of RAID, LVM and file system. An overview of the ZFS architecture is shown in Figure 2.3.

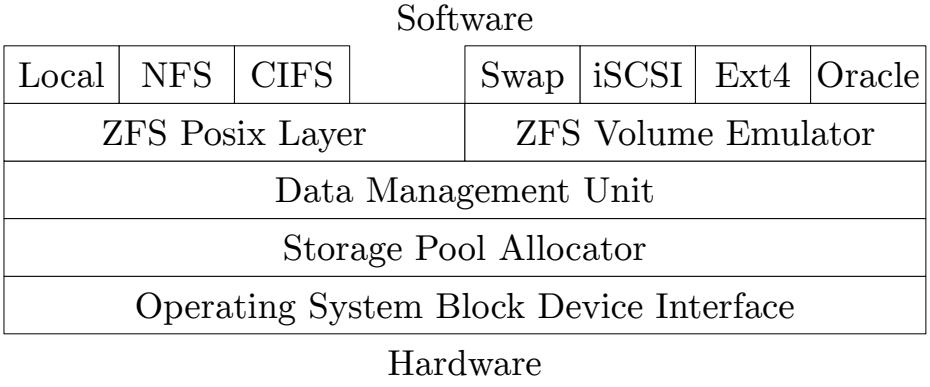

Figure 2.3: ZFS architecture

2.5.1 Storage Pool Allocator

Storage is organized into pools in ZFS. The Storage Pool Allocator (SPA) manages storage pools and virtual devices, allocating blocks at the request of upper layers. The designers wanted to provide simple block allocation, analogous to allocating RAM from the operating system. Because of this, ZFS uses a slab allocator [7], while many other file systems allocate blocks for files using extents [56]. Copy-on-write was seen as problematic for an extents-based allocator.

One system can have an arbitrary number of pools, but typically only one is used. A virtual device, or *vdev*, corresponds either to a physical device (or even a file) or a combination of physical devices [55]. These vdevs can be combined by mirroring or via RAID-Z (with one, two (RAID-Z2) or three (RAID-Z3) drives for parity). The hierarchy of vdevs forms a tree with the pool as the root. I/O is striped over the top vdevs in the pool. The SPA also handles compression and deduplication.

RAID-Z is similar to RAID 5 in that it uses parity for redundancy. ZFS avoids the read-modify-write procedure as it uses copy-on-write, where modified data is always written to an unused location in the pool. Unlike RAID 5, RAID-Z uses dynamic stripe width [23], so a small write does not have to be striped over all devices in the RAID-Z vdev. There is still some write amplification, since the parity needs to be written in order to guarantee redundancy. Similarly to standard RAID levels, mirroring delivers more IOPS

than RAID-Z.

2.5.2 Data Management Unit

The Data Management Unit (DMU) provides a transactional object store backed by the storage managed by the SPA. For organization, it is split into hierarchically named datasets that can inherit attributes and be managed as subtrees. Atomic transactions eliminate the write hole since a change is either committed or lost. ZFS provides hierarchical checksumming for everything that is written on disk, which ensures that not only is bitrot on disks detected, but also bitflips during transfer or corruption by failing disk controllers. The checksum for a block is stored in the block referencing that block, except for the *überblock* at the root of the tree, which contains its own checksum and is stored redundantly.

Rebuilding the data on a replaced device, or *resilvering*, can be faster with ZFS than conventional layered RAID. Since ZFS integrates RAID and file system roles, the information of which blocks are used is available when resilvering. Because of this, ZFS only has to resilver the blocks that are actually in use. The same applies to checking the consistency of the data stored in ZFS, or *scrubbing*.

When conventional RAID is scrubbed, the RAID controller can only tell which device is failing if the failure is noisy. If a device instead silently returns invalid data, conventional RAID can only notice that the data returned from devices does not match on a mirror or parity level. Thanks to checksumming, ZFS knows which device returns invalid data and can try to rebuild the data if sufficient copies or parity information is intact.

Since the checksum for a block is stored in the block referencing it, an update to a block leads to an update to its parents all the way to the überblock. Because ZFS uses copy-on-write, the transaction is not committed before the überblock is rewritten. The old data is still present since the new data is written in an unused location in the pool. When the überblock is rewritten, the old data becomes garbage and its location can be used for a new write. This makes it possible to implement atomic snapshots. Instead of letting the old data be rewritten because nothing references it, the state of a DMU dataset can be atomically snapshotted by storing a reference to it.

2.5.3 ZFS Interface Layer

The ZFS POSIX Layer (ZPL) implements a fully POSIX-compatible file system as a DMU dataset. File systems can grow until the pool runs out of free space. This can naturally be limited by quotas. Alternatively, a file

system can have a reservation so that other file systems in the pool cannot cause it to run out of space.

There are other layers implemented on top of the DMU, such as the ZFS Volume Emulator, which allows for creation of block devices called *zvols*. These zvols can be exported using iSCSI, used as block devices for other file systems, for applications that require raw block devices, or used as swap space. They offer some of the same features, like compression, checksums, snapshots and deduplication, which are implemented by lower layers.

ZFS management is performed primarily with two tools: *zfs*(8) and *zpool*(8). Pool management is handled by zpool: adding and removing pools, adding or replacing devices from pools, scrubbing etc. File systems are managed by zfs: adding, removing, snapshotting, sharing (via NFS or CIFS), diffing etc. The tools provide a much simpler interface than many contemporary file systems, which require changes to partitioning, RAID, LVM and file system to get similar tasks done.

2.5.4 Tradeoffs

Some of the features of ZFS come at a price. Copy-on-write can lead to similar fragmentation-related performance problems that log-structured file systems suffer from. ZFS has several performance focused design decisions that try to overcome these problems. Since writes are transactional, ZFS works with drives with enabled write cache, as long as the write cache correctly implements write barriers. This can provide much better performance than constantly flushing the write cache.

The designers of ZFS concluded that write performance dominates overall performance. Because of this, ZFS prioritizes writes. With a hardware setup that is capable of delivering some target write performance, a given read performance target may be achieved by adding caching. If the read patterns make caching unsuitable, the hardware has to be scaled up to meet said performance target.

ZFS uses a caching algorithm called Adaptive Replacement Cache (ARC). It can have a significant positive impact on performance, but requires lots of RAM to perform well. It is also possible to add a second-level cache (L2ARC), by adding faster block devices, like faster HDDs or SSDs. Transactions are only committed to disk periodically in order to limit the frequency of überblock rewrites. To guarantee software using *fsync*(2) that the data actually is committed to disk, the ZFS Intent Log (ZIL) is used. By default the ZIL resides on the data disks of the pool, but is normally only written to. A copy is kept in RAM and the ZIL on non-volatile media is only read when recovering from a crash. To speed up synchronous writes, a separate

synchronous log (SLOG) device can be added to the pool. In order to improve redundancy, there can even be several mirrored SLOG devices. A fast SSD works best for this purpose, as large capacity is not needed. This combined with copy-on-write allows ZFS to convert small random writes into large sequential writes.

There are several available compression algorithms in ZFS. LZ4 [12] is often recommended, as it is very light on resources while being fast and able to detect incompressible data. Often the I/O is the bottleneck and not CPU [62], in which case compression improves performance, as there is less I/O to perform.

While possible, it is not recommended to run ZFS on a 32-bit system. Many ZFS features make it much more RAM-intensive than other file systems. 64-bit systems are thus recommended for stability and also for sufficient RAM. The ARC can consume very large amounts of RAM. On Linux the effect is amplified, as the ARC does not share the page cache of other file systems and thus reacts differently to memory pressure. Deduplication, which saves space by only storing one copy of duplicate data, is a double-edged sword. It can improve performance much more than compression, by reducing I/O and improving cache hit rate, but increases RAM use further. ZFS deduplication is performed online, i.e. data is deduplicated while it is written, not by a scheduled deduplication task. In order to perform deduplication, ZFS keeps a table of hashes of deduplicated data blocks. The deduplication tables are stored in the pool with the file data and cached by the ARC, but prioritized in the cache as metadata. While the numbers are always dependent on the specific use case, the recommendation is to reserve multiple GiBs of RAM per TiB of deduplicated data.

The capacity of a ZFS pool can be increased by either replacing all drives in a vdev with larger drives or by adding new vdevs to the pool. Inconveniently, the capacity of a pool cannot be reduced. It would require block-pointer-rewriting, a complicated feature that is still in its infancy.

Copy-on-write can cause actively changing objects to fragment heavily. Fragmentation makes finding contiguous free space harder. This, in turn, causes ZFS to suffer more from fragmentation than many other file systems [24]. It is not recommended to fill a ZFS pool over 80% to prevent this fragmentation, but heavy fragmentation can cause significant performance degradation even in pools with a lower usage level [59].

Scrubbing and rebuilding a replaced device can take longer than with conventional RAID if the pool or device is very full, since ZFS has to traverse the data structures in order to find the checksums and data. If ZFS fails in an unpredicted way, there is no *fsck*(8)-like tool; there is only *zdb*(8), which is not designed for the same task.

2.6 Hadoop Distributed File System

Apache Hadoop is an open source software framework that was originally created to distribute and scale the storage and computations for the Nutch search engine [9]. The design is heavily influenced by the papers describing Google File System [17] and MapReduce [13]. Hadoop Distributed File System (HDFS) is, as the name implies, a distributed file system designed to scale using commodity hardware. HDFS was designed to handle amounts of data that previously required specialized and costly enterprise solutions by instead employing thousands of commodity computers. A HDFS cluster is comprised of nodes performing one of a handful of roles.

A *namenode* is responsible for the metadata of a namespace. Originally HDFS only allowed a single namespace, but this limit has since been removed. All of the metadata is kept in RAM in order to achieve desired performance. Changes to metadata are written to a transaction log on a local file system and periodically snapshotted into a file system image. A single active namenode simplifies the design, but it must only perform essential duties in order to scale and not become a bottleneck for the cluster. No writes or reads of data go through the namenode, only metadata requests to locate data or update information about it.

Datanodes are used to store the actual data blocks. They are usually equipped with a larger number of hard disk drives and less RAM than namenodes. Each drive typically has an individual operating system level file system on it without RAID or LVM. However, RAID can be used to improve performance via striping. Datanodes store blocks as files on the local file system, named after a block identifier, without any information about which file it belongs to in HDFS. Namenodes periodically receive heartbeats from datanodes, containing a report of the blocks a datanode possesses.

Clients connect to the namenode for metadata operations such as file lookup, creation, reading or appending. Mutual exclusion is handled per file by a writing client having to acquire a lease from the namenode and having to renew the lease in order to detect client failures. The namenode informs the client which datanode to contact for each block. Block size in conventional file systems is typically the size of a disk sector or a virtual memory page. In HDFS, blocks are much larger than on local file systems, so that the namenode can scale while keeping track of all blocks in RAM. The default block size is 128 MiB and typical values range from 64 MiB to 1 GiB.

HDFS is designed to withstand failure of drives, computers or entire racks, as a cluster with thousands of nodes is statistically going to experience very frequent single component failures. Each block is accompanied by a checksum

to detect silent data corruption. Data blocks can be replicated to several datanodes to provide redundancy in case of drive or node failures. The default replication factor is three copies for each block, with two going to nodes in the same rack and one going to a node in a different rack in order to protect against rack-wide failures such as network problems. The replication factor can be managed per file.

As stated, HDFS scalability is limited by the namenode because it has to keep track of all files and blocks in a namespace. The metadata per file or block is kept as small as possible, but as a rule of thumb, the namenode should have approximately 1 GiB of RAM per million blocks stored [53]. A file requires at least two block structures to be held in RAM on the namenode: one for the file entry and one for the first block of data. If the average file size is less than the block size, the resources of the namenode are used inefficiently compared to how much space it could handle.

With federation, HDFS made it possible for several namenodes to share the same datanodes. A single namenode still manages a single namespace and one application typically uses a single namespace. In addition to scaling the total number of files in a cluster, multiple namenodes isolate applications from each other, preventing one from congesting the namenode with metadata requests.

The namenode remains a single point of failure for a HDFS namespace. A hardware failure on the namenode affects the whole namespace, whereas datanodes are not a problem with sufficient redundancy. Namenode restarts may take a long time because the namenode has to rebuild the in-memory image of the namespace. The namenode does this by replaying the transaction log on top of the latest snapshotted file system image. A checkpoint node may be added to a namespace that periodically downloads the file system image and transaction log from the namenode, merges them locally and uploads the new image back to the namenode. Alternatively backup nodes synchronously maintain a copy of the namespace in memory. Both of these improve the namenode restart performance, reducing unavailability. A high availability namenode configuration is also possible by using an active and a standby namenode, sharing the snapshotted file system image and transaction log, and using a distributed decision making process to elect the active node in case of automatic failover.

While HDFS supports a wide variety of applications, it was originally designed to be used with MapReduce. Even though HDFS superficially resembles a POSIX file system by allowing hierarchical organization of files with owner, group and permissions etc. in directories, it is not, however, a POSIX-compatible file system. HDFS lacks, among others, *truncate*(2), random writes, soft links, hard links and POSIX locks. Files in HDFS could

originally not be appended, only written once and then read [9]. Both HDFS and MapReduce were originally designed for non-real time batch processing of large amounts of sequential data. MapReduce takes advantage of HDFS by running computations on the same computers that store the data, benefiting from local data transfer instead of sharing the available network bandwidth. Small files, random reads and writes, and real time performance were left out from the scope of the design of HDFS.

Both the original design and some of the limitations of HDFS mirror those of its architectural model, GFS. Namenodes are called masters in GFS and datanodes are referred to as chunkservers [17]. Some of the problems, like namenode scalability, were initially solved in a similar manner: by having several masters use the same chunkservers for storage while presenting different namespaces to clients. GFS has scaled further by introducing distributed masters, which allow for improved availability and solve the scalability limit of a single master per namespace [31]. HDFS has yet to catch up to GFS in this regard.

2.6.1 HDFS-RAID and Xorbas

One of the largest economical downsides of HDFS is the cost of redundancy. Each block is by default stored with a replication factor of three in order to protect against data loss and improve availability. In HDFS clusters storing petabytes, even a modest improvement in the effective replication factor can have a considerable financial impact. It should come as no surprise that this area has been under active research.

Facebook reported [60] to have implemented HDFS-RAID in order to save capacity in their HDFS clusters. Normal HDFS replication works similarly to RAID 1, storing several copies of the same data for redundancy. HDFS-RAID instead resembles RAID 5, as data plus parity are stored. Lost data can therefore be recovered from remaining data and parity.

The parity in HDFS-RAID is computed over a stripe of blocks as shown in Figure 2.4. There are two available encodings, XOR and Reed-Solomon [42]. XOR can only be used to compute a single parity block per stripe, protecting against the loss of any single data block. Reed-Solomon can be used to compute a given number of parity blocks, producing the desired level of redundancy. Facebook selected XOR using a stripe width of 10 blocks, with data blocks and parity blocks stored with a replication factor of 2. This resulted in an effective replication factor of 2.2 instead of the default 3. Another test was done with Reed-Solomon using the same stripe width, but a replication factor of 1 and 4 blocks of parity, resulting in an effective replication factor of 1.4. While XOR using these parameters would not result

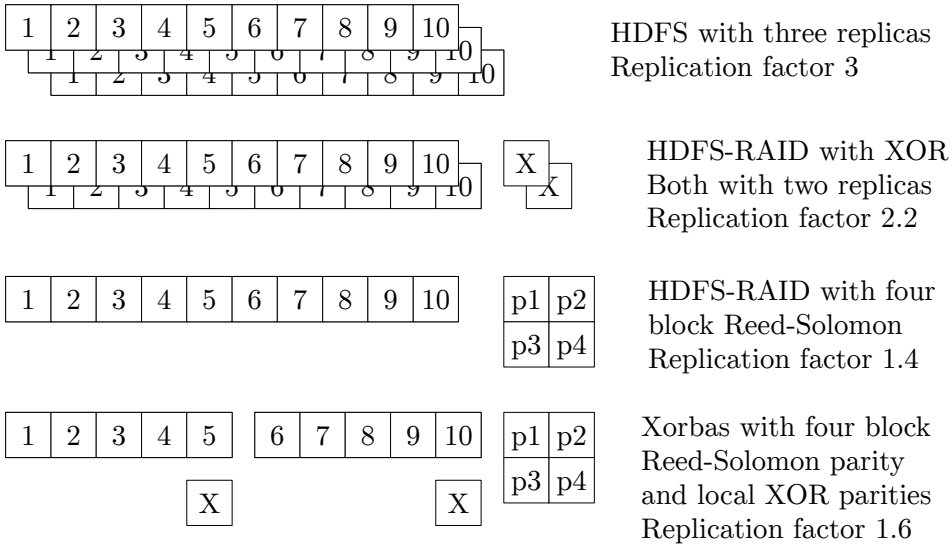

Figure 2.4: HDFS redundancy schemes

in data loss from the loss of any two blocks, the Reed-Solomon configuration would on the other hand be able to recover from the loss of up to four blocks.

HDFS-RAID is currently implemented asynchronously. Data is first written using normal HDFS replication and when a file has remained unmodified for a configurable period of time, parities are computed and stored, after which the replication factor of the original file is lowered. Since the parity is computed over a range of blocks per file, single block files cannot be parity encoded and have to remain replicated. Files shorter than the stripe width thus feature a higher than optimal replication factor.

In case of a corrupt or lost block, normal HDFS replication only has to make a new copy of a valid block. HDFS-RAID needs to read all the data blocks in the affected stripe, substituting parity blocks for the lost blocks, in order to recompute the lost data. This can cause a large amount of traffic in the cluster. Xorbas [45] is very similar to HDFS-RAID, but uses an alternative encoding scheme. A stripe is split into subgroups, each of which is used to compute a local parity block using XOR. Reed-Solomon parity is computed over the whole stripe. In case of the much more common single block failures, only the blocks of the subgroup have to be read to recompute the contents of the lost block, thus significantly reducing traffic. Rarer failures of several blocks can still be recovered within the limits of the redundancy of

the Reed-Solomon encoding. A very similar approach [26] is used in Windows Azure Storage.

2.7 Network File System

The Network File System (NFS) was originally developed by Sun Microsystems in 1984. It allows computers to access file systems located on other computers accessed via a network. NFS was designed to hide the fact that the file system is remote and instead operate as similarly to a local file system as possible. This allowed programs to take advantage of NFS with little or no changes and is usually referred to as POSIX semantics or POSIX-compatibility. As it gained in popularity, NFS was standardized as version 2 in 1989 [34] for interoperability between several vendors.

NFS was designed for local area networks with small latency and few transmission errors. The original translation of POSIX file access system calls to NFS remote procedure calls is simple, but each call introduces round-trip latency between the client and server. Often the program accessing NFS is performing a higher-level operation that results in several RPC operations and thus, additional latency. NFS version 3 was introduced in 1995 [10], featuring several improvements focused on latency. Asynchronous writes improved write performance while the addition of *READDIRPLUS* operation and changes to some existing RPC operations removed the need for separate calls in order to get file attributes.

Version 4 of NFS, originally published in 2000 [51], and revised in 2003 [52], was a much larger redesign. The original statelessness of the NFS server was abandoned and several auxiliary protocols were integrated into the main NFS protocol. While this complicated both the server and client implementation, it simplified deployment. Performance, specifically latency, was improved by introducing a *COMPOUND* operation [37], allowing several operations to be submitted in one request to limit cumulative latency.

While NFS is a very convenient solution for enabling network storage for programs not originally written with network storage in mind, it is not free of costs. Since NFS provides POSIX semantics, it has to support locks and other POSIX features that complicate caching. If the system that needs network storage does not need POSIX semantics, or can be redesigned to not need it, much simpler and more performant alternatives become available. HDFS not supporting POSIX simplifies its design tremendously. Facebook developed an image store called Haystack [6], which has even simpler design than HDFS.

Table 2.1 presents a comparison of ZFS, HDFS and NFS with currently common file systems such as *ext4* and XFS. This presentation is far from

exhaustive and is not suitable for picking the best option. Instead it is provided to demonstrate the wildly different design choices in the different file systems.

	ext4/XFS	ZFS	HDFS	NFS
Type	local	local	distributed	network
POSIX	yes	yes	no	yes
Checksums	no	yes	yes	depends[1]
Block size	typically 4 KiB	sector size – 128 KiB	typically 64 MiB – 1 GiB	1 MiB (TCP) or MTU (UDP)
Redundancy	depends[1]	internal RAID	replication factor	depends[1]
Consistency	journaling[2]	copy-on-write	depends[1]	depends[1]
Block allocator	extents	slab allocator	first local, then replicated	depends[1]

Table 2.1: File system differences

[1]Depends on the underlying layer or file system
[2]Journaling is usually only used for metadata. The data is updated in-place

2.8 Other Large Scale File Systems

Large scale data storage and processing is a very active and still growing field. It is not surprising that there are several alternative and competing technologies. Comparing all of the alternatives in depth is not the main focus of this Thesis. Some popular alternatives are covered briefly in order to demonstrate how similar they are to those covered in previous sections and to present some of the main differences.

BeeGFS is a distributed file system originally developed by Fraunhofer Institute for Industrial Mathematics under the name FhGFS. The research behind BeeGFS was spun off as a separate company called ThinkParQ [22]. The design is somewhat similar to HDFS in that it also has separate metadata and storage servers, allowing clients to bypass the metadata servers once the correct file has been located. In contrast to HDFS, BeeGFS has been designed to scale incrementally to multiple metadata servers by assigning each directory to a metadata server with available capacity. As a much younger project, BeeGFS has a smaller user base and lacks features such as checksums and parity-based redundancy.

Ceph [61] is a group of related distributed storage components. At the core is a distributed object storage system called Reliable Autonomic Distributed Object Store (RADOS). Similarly to HDFS, clients bypass the metadata servers and directly access the object storage servers once the required metadata has been retrieved. The namespace can be split between several metadata servers for scalability. Block devices can be provided for e.g. virtualization workloads by the RADOS Block Device service, which is implemented on top of the object storage layer. A file system with POSIX semantics, the Ceph File System [11], is also under development, but not currently considered production ready.

Lustre [28] is a POSIX-compatible distributed file system. Like in several other systems, storage is split into metadata servers and object storage servers. Lustre can support multiple metadata servers, which clients use to locate the correct objects and then proceed to access the objects directly on the object storage server. Files are striped over many objects stored on several object storage servers in order to improve performance. The POSIX-compatibility leads to a more complicated architecture compared to HDFS.

Gluster [40] is a distributed file system without a centralized metadata component. All Gluster nodes are aware of all peers, but instead of involving all nodes in every operation, the target nodes for an operation are determined by hashing the file or directory path and only the required nodes are engaged in the operation. A single file is not split into chunks by default but instead stored on specific file systems on the nodes, depending on the replication setup. Striping can be used, but is only recommended when files are too large to fit a single file system. The lack of global state in a cluster can lead to cluster partitions, which are hard to detect and recover from.

Cluster file systems such as Red Hat Global File System 2 (GFS2) [41] or Oracle Cluster File System 2 (OCFS2) [32] are designed to be used with shared storage. Nodes with access to the file system are connected via iSCSI or Fiber Channel to a Storage Area Network (SAN) hosting the actual storage capacity. Read operations typically perform well, as all nodes have direct access to the storage provided that the SAN does not limit performance. Write operators need to be coordinated among the nodes, which introduces latency and limits scalability. Both GFS2 and OCFS2 feature POSIX file system semantics.

IBM General Parallel File System (GPFS) [47] was originally a cluster file system that utilized shared storage. Version 3.5 added a feature called File Placement Optimizer (FPO) [29], which enabled GPFS to store data on the nodes accessing the data instead of in a SAN. The files are split into chunks and replicated much like in HDFS. GPFS offers distributed metadata and a POSIX-compliant file system in contrast to HDFS. GPFS with shared

storage has the same limitations as GFS2 and OCFS2. FPO is similar to HDFS, but less popular and not as proven.

Chapter 3

Environment

ZenRobotics was founded by Tuomas Lukka, Harri Valpola and Jufo Peltomaa in 2007. The founders wanted to turn their interest in artificial intelligence and robotics into commercial real world applications. After evaluating different potential applications the focus was set on automating sorting in waste recycling.

The ZenRobotics Recycler is an automated robotic waste sorting system that employs sensor fusion, computer vision, machine learning and artificial intelligence to identify and remove wood, metal and stone fractions from construction and demolition waste. Traditionally industrial robotics has focused on assembling or producing identical products from high-quality raw materials using a repeatable and precise process. Efficient robotic waste sorting requires reacting to sensor inputs in a chaotic environment and making intelligent decisions in real time.

The company has its own, very distinctive, culture. Since the product development requires significant amounts of expertise, employment is biased towards lots of research and software development experience. Most research and development personnel have years of experience, not just in their specific specialty, but also related technologies such as UNIX or differences of programming languages. There is a strong culture of challenging industry standard technology choices and solutions. After all, the current industry standard solutions for sorting waste do not make use of robotics or artificial intelligence.

3.1 High Level Data Processing

The ZenRobotics Recycler employs a wide range of sensors to measure properties of the incoming waste stream. All of the available sensory inputs are

combined and fed to a machine vision-based recognition system that produces a view of the objects in the waste stream. This information from one or more sensor arrays gets passed to a central node that decides which manipulators try to pick which objects.

The amount of raw sensor output is demanding. Each sensor is connected to a dedicated network interface card. In case something particularly interesting, such as an error, occurs, relevant data gets transferred to central storage at the office, as the installations at customer sites lack the resources to do thorough analysis. In case there is nothing of interest, a small amount gets sent anyway as a sample of normal sensor inputs.

Humans examine the interesting datasets and annotate the combined sensor outputs with object shapes and materials based on human vision and domain knowledge. These annotated datasets form the ground truth used to train the machine vision classifiers. Unannotated datasets are purged from storage when they are deemed expired after a set period of time.

3.2 Architecture

> Organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations
>
> M. Conway, 1968 National Symposium on Modular Programming

Different teams have been responsible for each of the subsystems of the ZenRobotics Recycler that produce data and analyze it. The teams continue to consume the data mostly in isolation from each other. A dataset can be defined, as per usage in the teams, as the data stored from a single run (from start to finish, or often, error). Datasets are immutable. They are written once, after which they do not change. Datasets are transferred and copied to various locations and removed when they are no longer of any use. Each team produces their own datasets in varying formats and quite different contents.

3.2.1 Sensors

The ZenRobotics Recycler relies on a wide variety of sensors to produce an accurate model of the incoming waste stream. The sensors include height map sensors, visible light sensors, near infrared sensors and metal detectors. Each sensor output is timestamped and stored as a separate data stream

called *channel*. The channels are preprocessed and compressed in various ways. Compression needs to be quick in order to keep up with the incoming data stream and to incur minimum latency on the processing pipeline. Most of the employed compression methods are lossy and the preprocessing also smoothes the data in order to reduce noise.

Compressed channels are sent to the recognition subsystem for further processing. In addition the channels are stored as *zensorserv* datasets in a ring buffer to be retrieved if needed. A small sample is sent to central storage for statistical analysis.

3.2.2 Recognition

The recognition subsystem does not produce any original data. A model representing the incoming waste stream is built in real time using the sensor data as input. Data from the height map sensors is used to construct shapes for the waste on the conveyor belt, while the metal detectors, visual spectrum-, and near infrared sensors are used to recognize different materials. The model is also segmented or split into separate waste objects in order to determine what can be picked up separately.

The machine learning-based system is trained with sensor data combined with annotations made by humans. Compressed inputs, even lossy, are used instead of raw sensor data in order to keep the process deterministic. The same result can always be reproduced using the same inputs and the same version of the code.

3.2.2.1 Vcache

Vcache is very simple persistent cache of costly computations. Many of the uses of Vcache are not really caching. In several cases the computation takes such a long time, that if the cached result is not available, the process fails in practice; a human annotator decides not to wait for the computation to finish.

Initially Vcache was backed by HBase [3], a distributed scalable data store built on top of Hadoop. As the amount of data grew, HBase was deemed unsuitable for storing the cache. Vcache was modified to use a file-based back-end called *Vcfs*. Vcfs stores the cache as files in two main directory hierarchies, as seen in Figure 3.1. One is called the index, where path components are analogous to function parameters. At the leaves of the index tree are small, 40 byte files that contain the hexadecimal representation of the SHA1 hash of the function result. The other directory tree is where the results are stored.

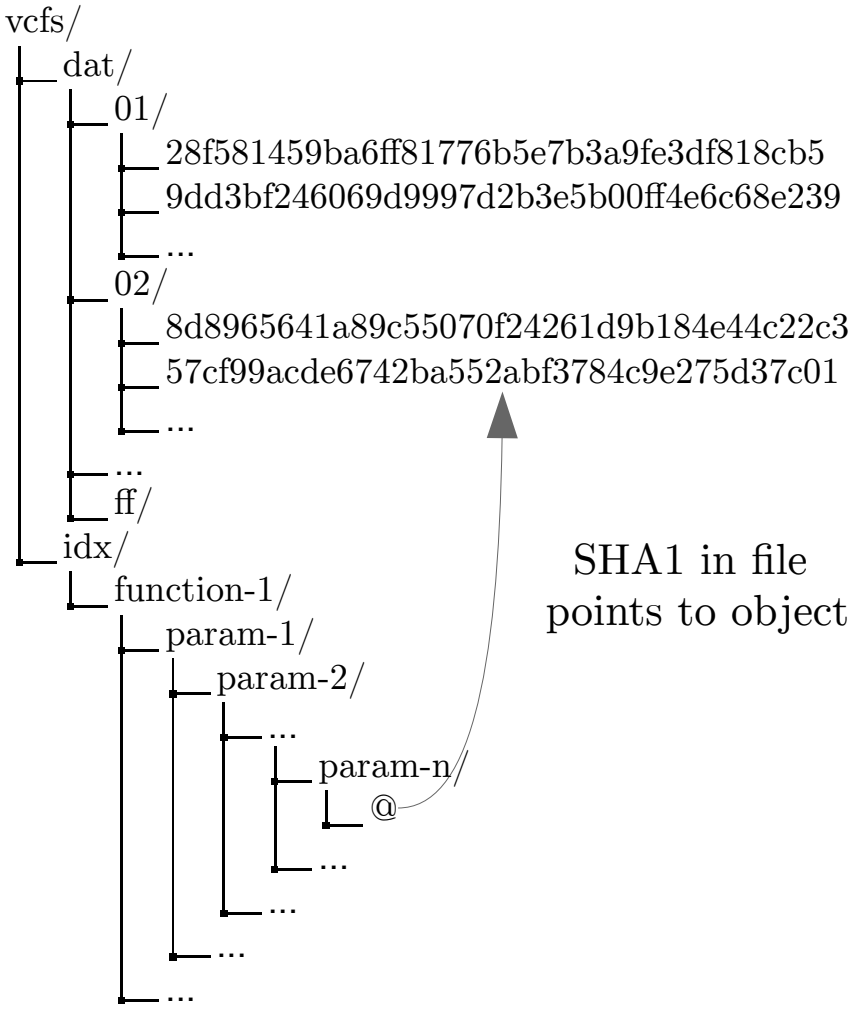

Figure 3.1: Vcfs second version structure

It is split up in 256 directories in the range 00–ff, corresponding to the first byte of the hash.

This layout allows for deduplication. Even if several computations produce the same result, it will only be stored once since the data is stored in a file named after the hash of the content. Unfortunately it uses a lot of inodes and it is very hard to perform reverse queries. Because of this, it is hard to selectively purge the cache.

In practice Vcfs is sharded on two Network Attached Storage (NAS) appliances (specification shown in Table 3.1) accessed over NFS. Performance

Manufacturer	Netgear
Model	ReadyNAS Pro 6
CPU	Intel Pentium E5300
RAM	4 GiB
HDDs	6x Seagate ST33000651AS [48] (4+2 RAID 6)
Connectivity	2x1Gbps Ethernet

Table 3.1: NAS specifications

is suboptimal for historical reasons. Vcfs shares the storage space with other, older data. The original requirements were for capacity over performance, causing RAID 6 to be used on the NAS appliances. Vcache causes mostly random reads while RAID 6 effectively limits the performance to single HDD IOPS levels.

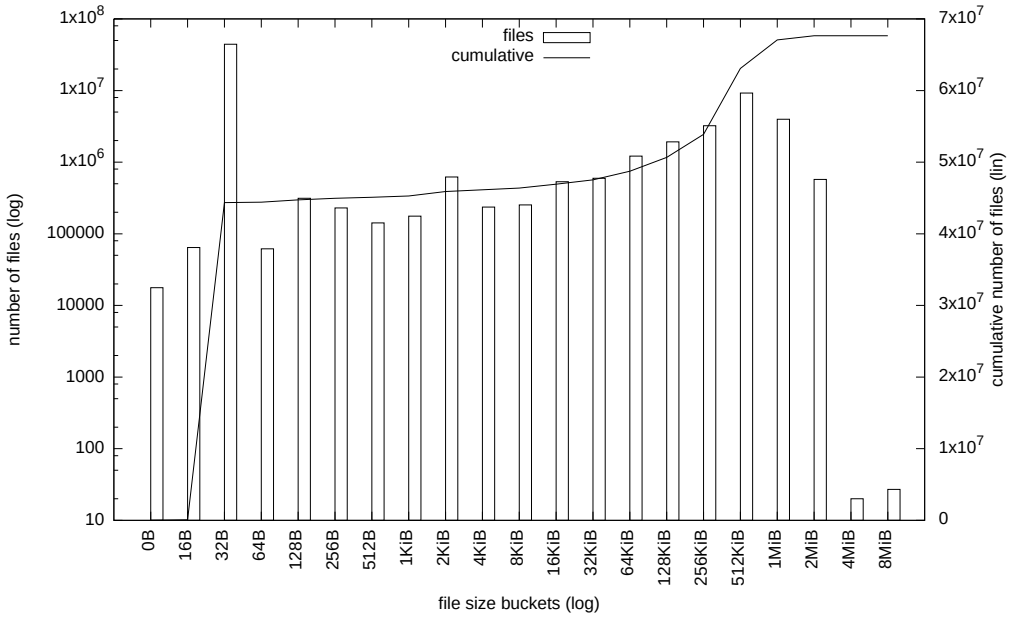

Figure 3.2: File counts in Vcfs

However, the number of used inodes is a bigger problem. The appliances use ext4 as the file system, which statically allocates the number of available inodes, and thus maximum number of files, during file system initialization. The default configuration of ext4 is to allocate one inode per 16384 bytes. If the average file is smaller than the space allocated per inode, the file system can end up not being able to create more files even though there is free space

to grow already existing files. Figure 3.2 shows the large number of small files, especially in the size range of 32 to 63 bytes, where the files in the index hierarchy storing 40 byte SHA1 sums fall.

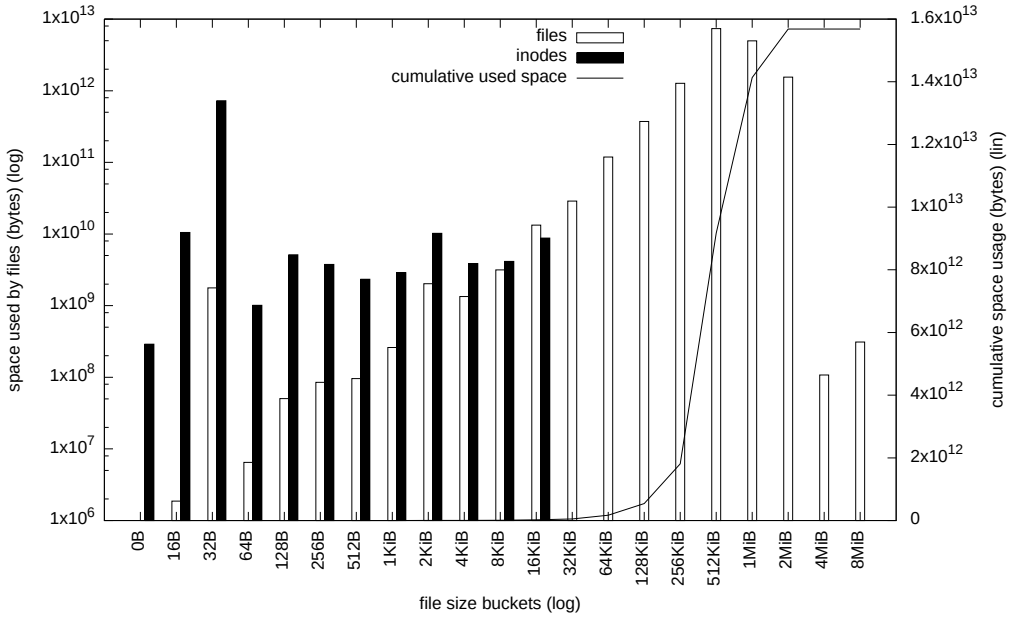

Figure 3.3: File sizes in Vcfs

The large amount of index files, each representing a key pointing to a value, does not necessarily translate to a large amount of raw data. In fact, as Figure 3.3 shows, the contents of the index files only take up a few gigabytes. However, storing such small files is wasteful and with the space allocation per inode overhead, the index allocates three orders of magnitude more storage capacity.

The impact of these problems can be mitigated to some degree. If re-initialization of the file system is a possibility, the inode allocation can be increased to one inode per 4096 bytes. This would maximize the amount of inodes, since ext4 allocates space in 4 KiB blocks. The file system would not run out of inodes before all blocks are also in use.

Performance of Vcfs would also improve by increasing the number of IOPS the NAS appliances can perform. Again, if re-initialization of the file system is possible, the RAID level could be changed from RAID 6 to RAID 10. The available storage capacity would decrease a bit, but increasing the number of inodes would help, as inodes are scarcer than space. The IOPS performance could be up to three times that of the current, significantly improving the

latency of looking up an index file or corresponding object file in the file system hierarchy.

Some Vcfs problems are directly attributable to the design. While the desire to deduplicate identical results is understandable, it doubles the inode usage. Another way to achieve the same result would have been to hard link the index file to the file in the *dat* hierarchy, instead of storing the SHA1 hash in the index file. A file in the *dat* hierarchy with a link count of only one could be determined to have no keys pointing to it and thus be garbage collected. While ext4 does have a limit for the number of hard links, the limit is quite high at 65000. Other software, e.g. BackupPC [5] that uses hard links for deduplication, have found solutions for working around the limitation.

While using hard links would decrease the number of inodes, it still would not help with the large number of disk seeks needed to traverse the directory hierarchy in order to find a specific file. Using NFS also increases the latency of directory traversal and file lookup. The file system layout of Vcfs also does not allow for efficient reverse lookups in order to determine if an object is no longer referenced and can be garbage collected. Any removal of objects based on age would require walking through the complete file hierarchy. While age is far from a perfect cache invalidation policy, it is better than nothing. Implementing a LRU policy for Vcfs would require storing access times. Storing the access time metadata in the NFS inodes would transform every read to a write, further impacting performance.

Implementing a simple key-value store using only a POSIX file system and its metadata as the data structures imposes certain tradeoffs. The desire of simplicity, familiarity and easy debugging using standard UNIX tools is understandable, but actual databases have real advantages over file systems and should be seriously considered.

While migrating all 16 TB of Vcache promptly to a database is infeasible, most of the advantages can be claimed by migrating the index. The index contains above 44 million entries pointing to over 23 million objects. Deduplication is clearly effective. The hexadecimal representations of SHA1 hashes occupy 40 bytes each, totaling under 2 GB. However, that number does not account for the cost of storing the key, the file name, and directory metadata used to locate the hash. The average path length under the index directory is 457 characters. The keys and hashes can all fit in under 22 GB, a very manageable size for a database.

A database could be used to improve more than performance. If the database were to store the age of a cached object in addition to its location, that information could be used for cache invalidation instead of traversing the file system for file creation or modification times. Additionally, the key and hash would not have to be stored serialized as something that is safe

for a file system, but could be stored more compactly and efficiently, further improving resource requirements and performance.

3.2.3 Adaptive Picking

The adaptive picking subsystem is closely related to manipulation subsystem. Different grippers, object shapes and sizes require different alignments and positioning in order to succeed. Starting from very simple heuristics, the system to a large extent has taught itself to pick. The system has attempted different variations, storing the specific parameters, and sensors or human annotators going through the information have noted which attempts have been successful. Machine learning has been used to improve the adaptive picking performance until it was sufficiently good.

3.2.4 Manipulation

The manipulation subsystem is responsible for controlling the gripper and servos in order to pick objects up from the waste stream. While running, it produces a large amount of diagnostic information that is stored. Some of the stored data are inputs to the manipulation subsystem produced by other parts of the recycler, but stored by the manipulation system for easier correlation.

Previously the manipulation subsystem produced datasets that consisted of a very large amount of files in a directory. Some stored data streams resulted in several files per second, e.g. video where each frame was saved in a separate file. A lot of data was stored in human-readable formats such as JSON or Clojure symbolic expressions, which are not very compact representations.

Currently the system utilizes Robot Operating System (ROS) [39], which provides a far superior file format called ROS bags. Datasets stored as ROS bags multiplex several channels, or topics, to a single file. This way temporally related data is stored spatially colocated in the single byte stream and thus read-ahead caching performs well. ROS bags also support optional compression.

Some data streams are stored inefficiently due to legacy reasons. A bitmap representation of the work area of the conveyor belt is stored frequently, even though the conveyor belt advances slowly and each bitmap is mostly made up of an offset copy of the last bitmap.

The manipulation datasets also contain some of the most refined information, as the manipulation subsystem is the consumer of most of the information produced by the rest of the system. Many of the inputs are also combined and correlated by the manipulation subsystem. Therefore it makes sense to

store a wide variety of data in the manipulation datasets instead of elsewhere in the system.

3.3 ZenRobotics Dataset Storage

The current iteration (spring 2013) of dataset storage at ZenRobotics is mostly based on scaling up the previous solution. Our developers were accessing a Linux-based NAS, as specified in Table 3.1 over NFS, which was backed up to a second identical NAS. It was decided to keep the dual setup for redundancy and NFS since it had served well so far, thus not complicating the basic setup. Naturally, financial cost was also a factor.

There were already problems with backing up the several terabytes of content from the primary NAS to the secondary using *rsync* [58], since the number of files was quite high. ZFS was selected as the file system as it provided strong integrity guarantees, easy administration and was designed for large amounts of data.

The 10^{-14} non-recoverable bit read error rate of the Seagate ST3000NC002 HDD [50] suggested that reading once through the 30 drives ($30 * 3 * 8 * 10^{12} = 7.2 * 10^{14}$ bits) had a large chance of returning bad data without ZFS. Ext4 would have required us to split the storage on one server into several volumes or increase the space used by small files from the default 4 KiB. LVM would have meant tedious resizing of file system to shift around capacity. Traditional RAID would have meant that resilvering when replacing a failed drive would write the whole device even if it was not fully used. ZFS send and receive solved the problem of rsync slowing with the number of files. Compression was also enabled in ZFS once it was realized that is was essentially free, as the file servers were not constrained by CPU, but rather by I/O.

Having decided on ZFS, the hardware (Table 3.2) for it was selected. A relatively powerful CPUs for a file server was chosen, since ZFS calculates checksums for everything it reads and writes; large amounts of ECC RAM to benefit the ARC without compromising integrity; 1 TB of SSD capacity split over 4 drives for L2ARC; a small, but fast SSD SLOG to accelerate synchronous writes; 30 large capacity HDDs configured in three RAID-Z2 vdevs plus a spare. The secondary file server has identical hardware, to be used as spare parts if needed.

The decision was not without its drawbacks. ZFS on Linux was selected due to more comprehensive hardware compatibility compared to *illumos* [27], the Open Source Solaris descendant. ZFS on Linux has had some problems, but never risked the data. Using RAID-Z, even with three vdevs and a large cache, has a clear performance impact in concurrent use. ZFS performance degrades

significantly as the pool fills up and is fragmented. A scrub operation on the file server usually completes in under 24 hours, but with heavy concurrent access or a full and fragmented pool it has taken over a week.

CPU	2x 6-core Intel Xeon E5-2630
Motherboard	SuperMicro X9DRH-ITF
RAM	128 GiB (8x 16 GiB ECC)
HDDs	31x Seagate ST3000NC002 [50] (3x(8+2 RAID-Z2) + spare)
L2ARC	4x 256 GB Samsung 840 Pro SSD
SLOG	100 GB Seagate Pulsar.2 SSD
Connectivity	2x10Gbps Ethernet

Table 3.2: File server configuration

3.4 Amount of Data

So far ZenRobotics has managed to avoid completely filling up the current file server. Unfortunately, ZFS starts to fragment heavily as the pool grows near full, impacting performance. More aggressive data removal was required and storage quotas were enabled to prevent filling up storage. It is an unfortunate situation as more data is usually beneficial for data analysis. Additionally, spending human resources on deciding what can be removed and what must be kept has a negative impact on other work.

Conditions vary from installation to installation. Each installation needs the machine learning algorithms to be trained with data from that particular installation in order to perform well. Depending on various factors, an installation may produce 1–60 GiB of data per hour. Practically speaking, the amount of data being transferred from one installation to central storage is limited by Internet connection speeds.

It is difficult to predict future storage capacity needs. Less aggressive removal would already cause the file server to fill up. More aggressive removal may be feasible. It would be preferable to keep data for an extended period of time in order to have data produced under different seasonal weather conditions. The number of annotated datasets continues to grow. It is very unlikely that annotated datasets would be removed. The current data corpus has been produced from only a few installations. These installations provide an incomplete foresight, as they are routinely affected by research and development activities. Overall, storage use per installation will keep growing and each new installation will require a significant addition to storage capacity.

The storage usage varies constantly, as new data arrives and old data gets purged. Here is represented a point-in-time summary of the space usage by data from each subsystem:

- 23 TiB, Compressed sensor data, moderate number of files
- 9.8 TiB, New manipulation data (ROS bags), low number of files
- 5.1 TiB, Old format manipulation data, large number of small files
- 3.1 TiB, Miscellaneous research files, large number of small files
- 1.1 TiB, Adaptive picking data, moderate number of files

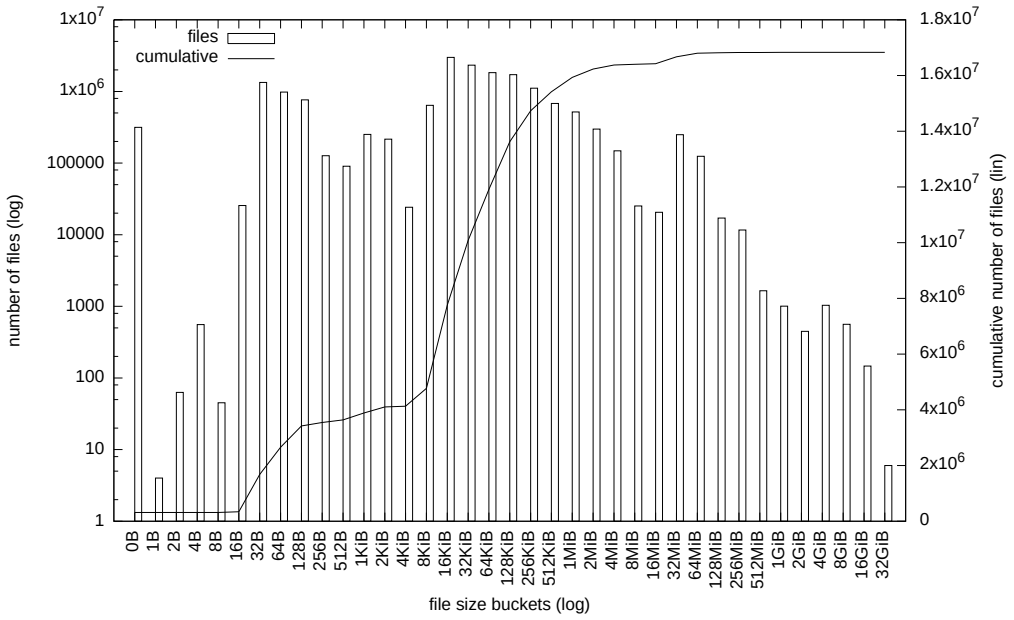

Figure 3.4: File counts

The old manipulation and zensorserv datasets use the presence of zero-length flag-files to indicate status of the datasets. Figure 3.4 shows how this contributes to the large number of zero sized files. The distribution also shows the large number of files in size ranges from 8 KiB to 1 MiB. As said, these largely correspond to single frame captures from video streams and similar uses.

Storage capacity is still dominated by large files, as shown in Figure 3.5. The largest files are mostly ROS bag files. Zensorserv dataset channels are

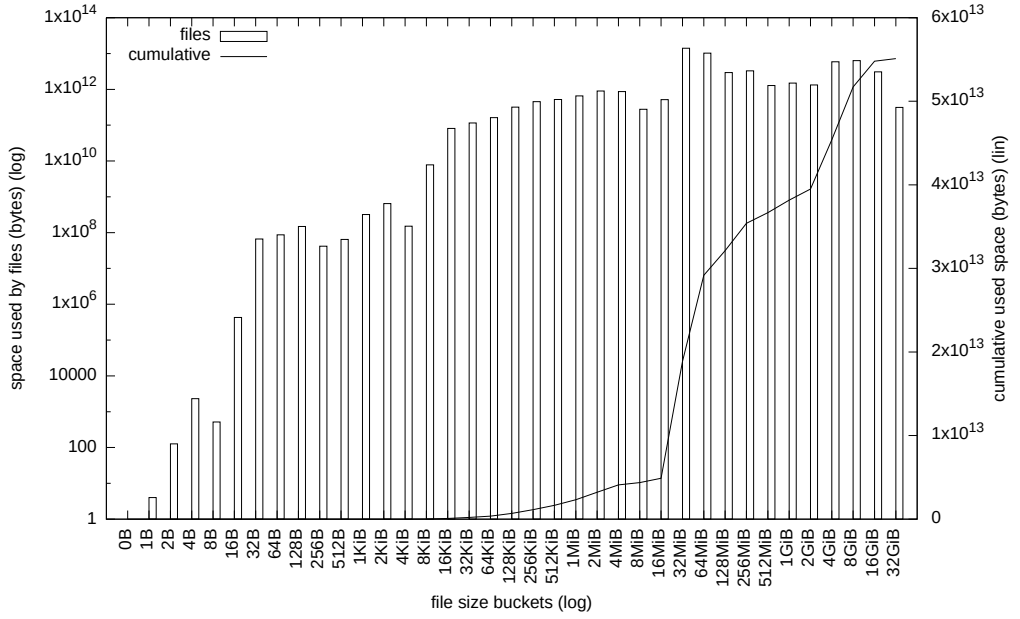

Figure 3.5: File sizes

split into files ranging from tens to hundreds of megabytes in size. The smaller files that overwhelm others in numbers in Figure 3.4 do not really contribute much to overall space usage.

The distribution of files per directory in Figure 3.6 features a few notable things. First, there are a large number of empty directories and directories with only one file. These can easily be eliminated. There are also a significant number of directories with large numbers of files. Such directories are mostly made up of video streams stored as single frame images.

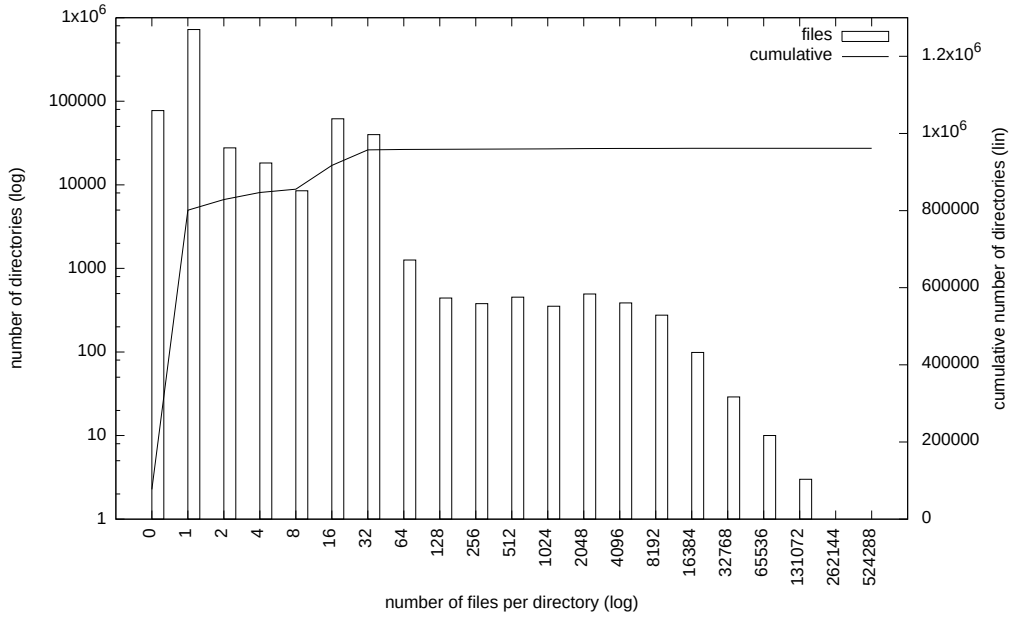

Figure 3.6: Directory sizes

3.5 Storage Utilization by File Type

As Figure 3.7 shows, the largest space savings would have to target .blob files in zensorserv datasets. The files are already preprocessed and compressed with specialized algorithms that have lot of domain knowledge about the data. Significant improvement in compression ratio seems unlikely. ZFS LZ4 compression shows a compression factor of 1.14 for zensorserv datasets, most of the savings coming from the non-.blob files in the datasets.

The old manipulation dataset format compresses quite well, even with a quick compression algorithm such as LZ4. ZFS reports a compression ratio of 1.62. This is mostly due to the well compressing human-readable files, like JSON logs, in the datasets. Unfortunately no single file type used by the old dataset format makes up a significant portion of the allocated storage.

ZFS also reports a compression factor of 1.47 for the new format manipulation datasets, stored as ROS bags. These .bags also make up a large portion of the storage allocation by file type. As mentioned earlier, ROS bags feature optional compression using gzip or bzip2. While these algorithms use more CPU cycles per byte than LZ4, they also usually result in better compression. Compressing a bag of several tens of gigabytes offline can introduce significant unwanted latency in the processing pipeline. It would be best to compress

the stream when the bag is originally produced. Compression is rarely used, as it slows down interactive exploration of the dataset. This issue may be improvable in the tools. Alternatively the tools could be modified to support faster compression algorithms, such as LZ4.

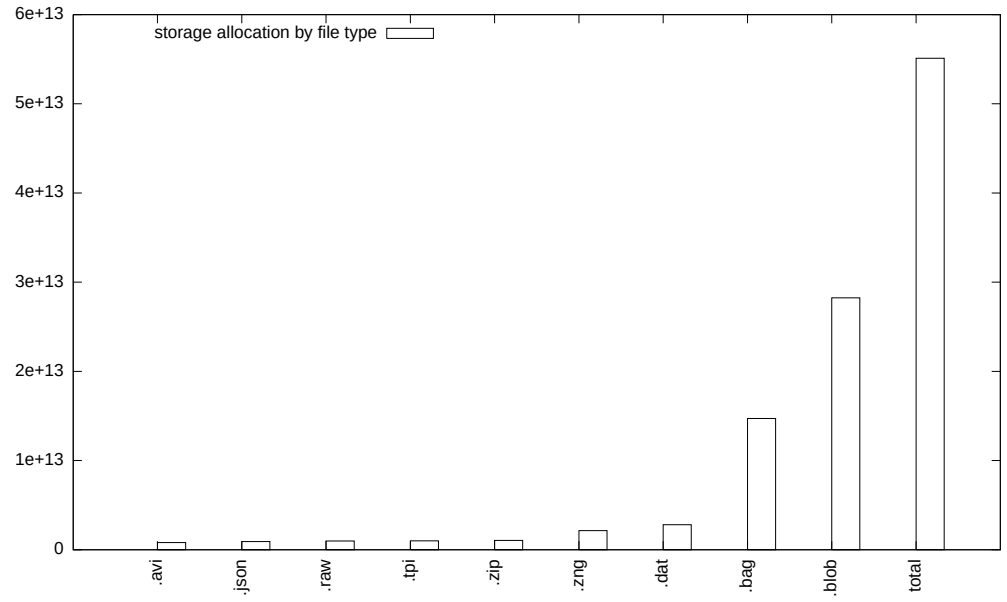

Figure 3.7: Top 10 File Types by Storage Utilization

Chapter 4

Experiments

While statistics were gathered from production systems to gain an understanding of the real world situation, tests were not performed in the production environment to prevent any effect to the daily usage and to isolate the tests. The tests were performed on surplus hardware with specification shown in Table 4.1, which could be dedicated to the task and configured much more freely according to the requirements.

CPU	4-core Intel Xeon X3430
Motherboard	SuperMicro X8SIA
RAM	8GB (2x 4GB ECC)
HDDs	2x Seagate Constellation ES ST1000NM0011 [49]
L2ARC	80GB Intel X25-M G2 SSD
Connectivity	2x1Gbps Ethernet

Table 4.1: Test server configuration

Not running tests on production hardware or hardware with the same exact configuration limits the options for testing. The tests were selected in order to understand fundamental properties of the technologies in use in the production system. Since the production storage system is based on HDDs that are used to store lots of small files, it is valuable to understand how HDD performance is affected by read/write operation size. Large scale file systems are carefully designed to keep the metadata per file to a minimum. ZFS is currently used at ZenRobotics, which leads to questions about the amount of metadata per file in ZFS. As datasets are accessed over NFS, we are interested in how NFS performance is affected by small file sizes. In Section 3.5 we looked briefly into how effective compression can be for dataset storage. All the compression algorithms were generic, with no domain-specific tuning. Therefore it needs to be determined, how much improvement

a domain-specific algorithm would provide.

4.1 I/O Read Operation Size Effect on HDD Throughput

In order to verify the simple HDD performance model presented in Equation 2.1, we performed a test on real hardware. A sufficiently large number of read requests were performed using each of the various request sizes to find out the effect on performance. Each read was issued to a random location on the HDD to minimize the effect of read-ahead and caching. Locations were sector-aligned to eliminate possible alignment overhead. The total time for a sequence of read requests using a single request size was measured. Given the amount and size of the requests and the time elapsed, we can calculate the throughput of the HDD.

To measure performance, one must try to isolate the test subject from everything unrelated. Since we wanted to measure the performance of the HDD and not the file system, we read from the raw block device instead of a file. Also caches had to be disabled or emptied to minimize their effect on the test.

The source code for the short test program written in Python can be found in Appendix A. It is organized in two phases. First all global initialization is done, including opening the HDD block device for reading. Then different block sizes are used to perform the test. Before each test, caches are dropped to minimize the impact of previous tests. Then the given size of block is read from a sufficiently large number of random locations on the HDD. The time taken to perform the reads is recorded. Based on this information, the throughput can be calculated for each block size.

Preliminary test results formed a curve shaped according to the model's prediction, but did not match it exactly, as can be seen in Figure 4.1. Firstly, the model predicted much higher maximum throughput. The parameter used for the model's sustained transfer rate was taken from the technical specifications of the HDD that was used for the test. We initially overlooked the fact that the specifications listed the maximum sustained transfer rate only on the outer diameter of the platter. The test reads from random offsets on the HDD, which span all of the tracks instead of the best performing outer track. We then measured the average sustained transfer rate of the whole disk by executing *dd*(1) with a block size of 1GiB for the whole block device. That gave us the average sustained transfer rate of 115MiB/s instead of the maximum sustained transfer rate of 147MiB/s. This average sustained

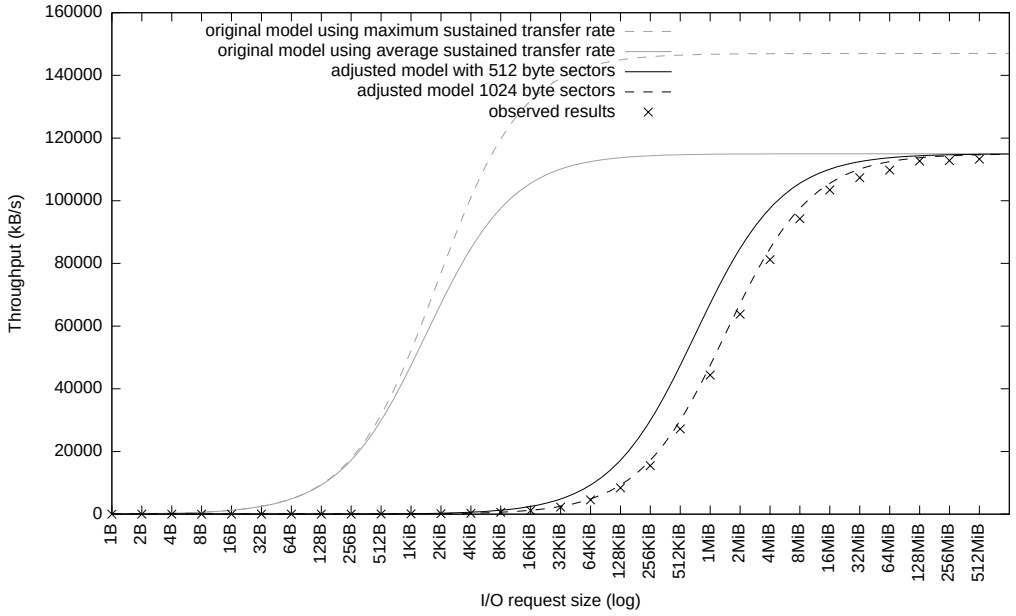

Figure 4.1: Throughput as a function of I/O operation size

transfer rate is almost perfectly in line with the maximum throughput of our test results.

$$R_{\text{avg}}(l_{\text{block}}) = \max\left(1, \frac{l_{\text{block}}}{l_{\text{sector}}}\right) \times \frac{1}{t_{\text{seek}} + \frac{\max\left(1, \frac{l_{\text{block}}}{l_{\text{sector}}}\right)}{R_{\text{sus}}}} \tag{4.1}$$

Secondly, we found that the model predicted performance to improve with a much smaller block size than what we observed. The sector size for this particular HDD model is 512 bytes. All reads smaller than a sector will still result in reading the whole sector. The model originally failed to account for this. The adjusted model in Equation 4.1 using a sector size l_{sector} of 512 bytes comes very close to predicting the results. The ability of the model to predict the results is even better with 1024 byte sectors, which leads us to believe that the HDD always does a one sector read-ahead for single sector reads.

4.2 ZFS Metadata Overhead

In order to examine the metadata overhead per file in ZFS, we performed an experiment in which we measured the amount of free space on a file system

before and after creating a number of files of a certain size. The test was repeated with different sizes and file counts. Given the size and number of files, we were able to subtract what was allocated by the files from the reported disk usage. The remaining difference in free space accounts for metadata.

ZFS has many features that make measuring the used space tricky. Free space counters only update periodically when transaction groups are committed, so that they can properly reflect space savings from deduplication and compression. Deduplication and compression must be disabled in order for them not to affect the reported usage. To force a transaction group commit, we exported the entire pool that held our test file system.

During initial prototyping, we noticed that a newly created file system did have more than zero bytes in use. After removing all files from the file system in preparation for the next run, used space differed from that of a newly created file system. Because of this we created a new file system before each run and destroyed it after collecting data.

A newly created file system also had varying amounts of used space depending on the pool configuration. Because of this, we ran the test both with a two disk striped pool and a 4 disk RAID-Z pool. The source code listing for the experiment can be found in Appendix B.

For each number of files of a certain size our test procedure produced a number of bytes used. We also measured the number of bytes used on an empty file system; 30720 bytes in case of the two disk striped pool and 42896 bytes in case of the four-disk RAID-Z pool. We subtracted the usage of the empty file system from the results. In the case of the 10 zero-length files, this led to zero space usage. We assume that an empty directory occupies a DMU record that can accommodate a number of file entries before allocating a larger record. In addition to subtracting the empty space, we subtracted the space needed to store the file names, as those were dictated by the test procedure and not the file system.

Figure 4.2 presents the test results. The X-axis shows the different test cases performed. The zero-length files can be seen to have very similar metadata overhead regardless of the pool configuration. This is to be expected as the only storage need is for the directory entries. The 1 KiB files have wildly different overhead on the 2-disk and 4-disk pool. The reason is the behavior of RAID-Z. For each written block, RAID-Z must write a parity block for redundancy. We are in fact observing RAID-Z overhead instead of metadata overhead. This is a situation where RAID-Z is less efficient than traditional RAID 5, but it is the cost of avoiding the read-modify-write procedure and the RAID write hole problem. The overhead should not be flatly dismissed as being caused by RAID-Z. It should be kept in mind and

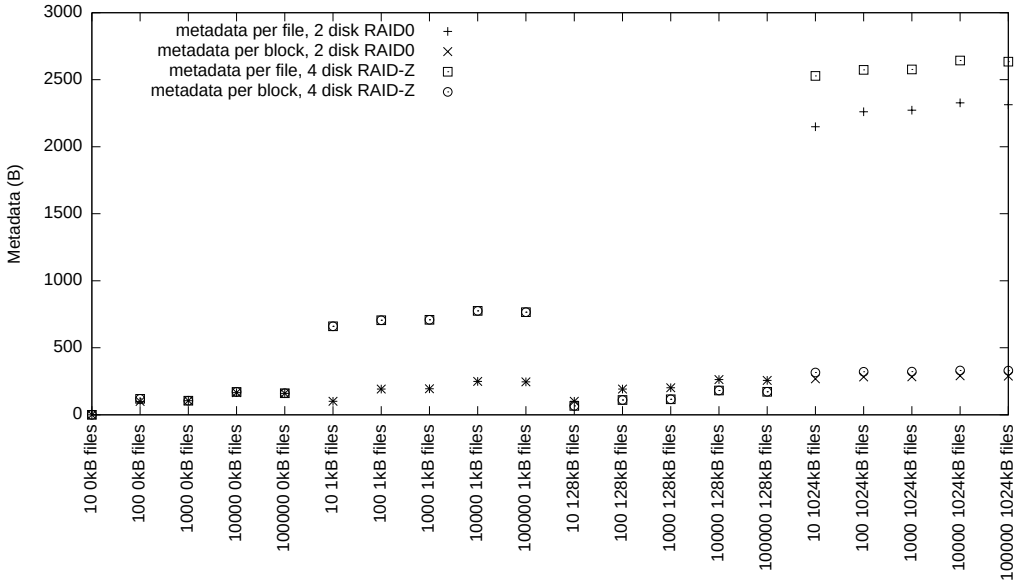

Figure 4.2: ZFS Metadata Overhead

noted that it is caused by the size of the write operations, instead of the size of the file, as one could be lead to believe.

At the size of 128 KiB we again observe the overhead in both the 2-disk and 4-disk pools to be very similar, as is the average of test cases up to this size. A larger file size averages out the differences in sector allocation between striping and RAID-Z. The 1MiB files break the pattern by showing much higher overhead compared to all other test cases. The DMU uses blocks, called records, of up to 128KiB. The content of all the smaller files fits in one DMU record. Since the larger files comprise several records, the overhead is the sum of the overheads of the records. If we divide the total amount of metadata with the number of records instead of number of files, we get results that are in line with the rest.

4.3 NFS Latency Overhead

To gain visibility into the impact of latency on performance, we performed a test on actual hardware. The intention was specifically not to compare bandwidth, but rather NFS RPC operation latency overhead. We measured the time taken for certain operations on a local file system and over NFS. Some of the operations only inspect the file metadata, while others also read

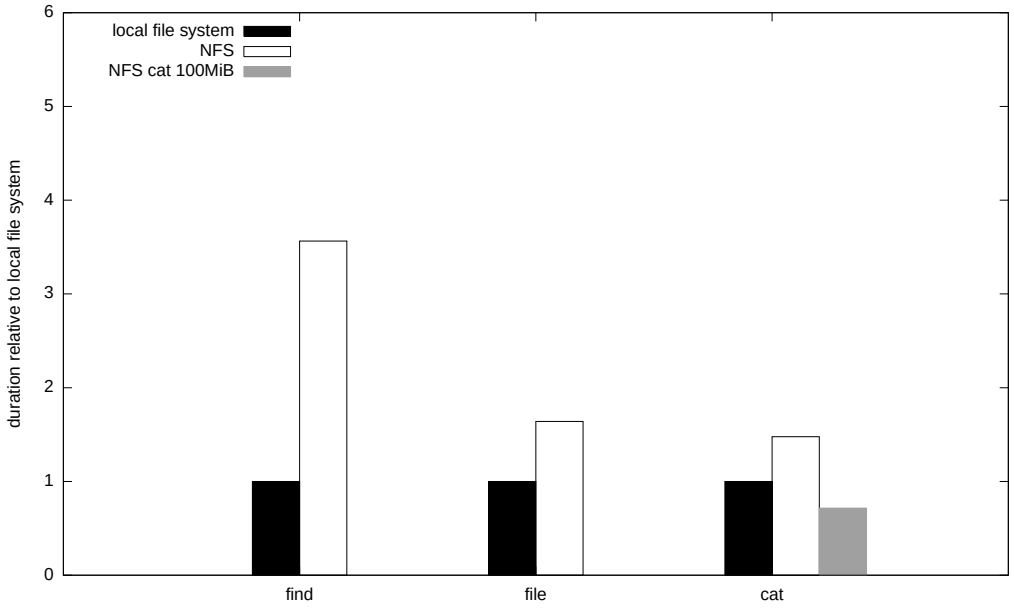

Figure 4.3: NFS vs. local file system, normalized relative to local performance

file contents. This will show if there is a difference in performance overhead for metadata vs. data operations.

A directory tree consisting of over 9000 directories, 2.2 million files and consuming 1.7 TiB of storage capacity was used for the test. The directories contained varying numbers of files of varying sizes and contents. One computer stores the files locally and exports them to another via NFS over a 1Gbps Ethernet LAN. Tests were run both locally on the machine storing the files and on the other accessing the files over NFS to compare the results. The commands used are listed in Appendix C.

First we measured the time elapsed for the directories to be iterated through by `find -ls` with output directed to `/dev/null`. This had the effect of reading all directory entries and also accessing each file's metadata stored in its inode.

The second measurement was for the time taken to iterate through the same files with `find -exec file` with output again directed to `/dev/null`. In addition to reading the metadata of each file, *file*(1) reads the first 96KiB of each file's contents in order to try to determine the format of the file.

Finally, we tested file reading performance. A list of files between 900kB and 1000kB was first prepared, in order to avoid spending time reading metadata. Then we measured the time taken to read the listed files using

cat(1). The total amount of data was 7.5GiB in 8407 files. Additionally, we wanted to demonstrate the effect of larger files. Since large reads would cause the local HDD bandwidth to dominate the 1Gbps Ethernet connection, it makes little sense to compare local and NFS performance. Instead we split the 7.5GiB contents of the files into 100MiB files and measured the time to read them over NFS.

The results shown in Figure 4.3 are scaled relative to the time taken by the test on the local file system. The find test, which consists purely of metadata reads, is comparatively slower over NFS than the file test. The cat test is, in turn, a small improvement compared to the file test, as 900-1000KiB are read per file, instead of up to 96KiB. The test of reading the same total data split into a smaller number of larger files is illuminating. It executes in half the time compared to reading from the smaller files, doubling throughput. All of these results point to the latency caused by metadata operations degrading overall performance.

4.4 Recompressing ZNG to PNG

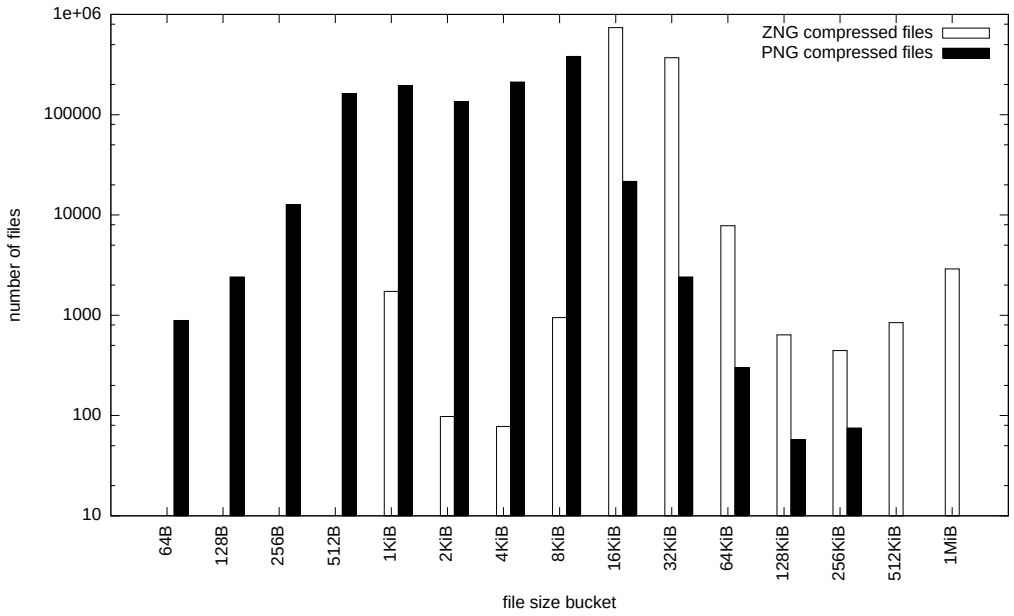

Figure 4.4: ZNG and PNG file size distribution

In Section 3.5 we studied storage utilization by file type. It is clear that the largest file types can be compressed more effectively than they are

now. But the fourth largest file type in Figure 3.7, the .zng. The files are compressed with snappy [18], without any domain specific knowledge. These .zng files are bitmap images for which there exist widely available, lossless and optimized compression methods and file formats, such as Portable Network Graphics (PNG). Snappy was originally chosen over PNG because it is less resource-intensive and CPU resources were constrained.

Conversion to PNG using the reference PNG implementation libpng [46] results in quite impressively improved compression ratio, as can be seen in Figure 4.4. 40 GiB of .zng files were converted to 6 GiB of PNGs. The files are quite small, average size being 37KiB for .zng and 5.6KiB for PNG. Often the files are smaller than the file system block size and thus take unnecessarily much space. Unfortunately they also make up only a small part of the total space used, limiting the impact of this improvement in compression has on the total amount of data.

Chapter 5

Discussion

The growing need for storage capacity at ZenRobotics will run into the limits of the current storage architecture. While the current architecture can still be grown, costs per TiB will rise. RAID 10 is needed for performance, leading to less usable storage per disk. Additional hardware is needed for housing more disks. At some point a single server will no longer scale. Instead of building logic for accessing multiple NFS servers, a different approach should be taken.

Current dataset storage usage patterns include a large number of small files, immutable datasets, random access, and bulk analysis operations. For large amounts of bulk data that has to be randomly accessible, hard disk drives are still the most economic option. The data is spread into many small files — sometimes, even zero length files that are used to indicate a one-bit status. Our experiments show that HDDs perform poorly with small read and write operations. Merely archiving the small files making up a dataset into a large file will not help if the usage pattern still consists of small read/write operations.

Using NFS for accessing the files also presents problems. The results of our experiments indicate that NFS operations incur latency that has a significant impact on throughput. NFS performance benefits from larger files much in the same way as HDD performance does.

Vcache performance impacts interactive use by annotators. The design magnifies the problems with small files, NFS latency and limited IOPS performance of RAID 6. Maintenance is also seriously hindered by the performance of iterating through cache contents. The Vcache index would benefit significantly from using a database instead of file system hierarchy over NFS. The use of NFS for Vcache objects should also be reconsidered. A non-POSIX object store would cover the requirements equivalently.

IO is often the bottleneck. CPU performance can be traded for IO performance by using compression; especially as CPUs have more and more

cores. ZFS LZ4 compression is currently used when storing datasets, which results in some space savings with negligible costs. However, file contents are not compressed when transferred over NFS. According to Nielsen [33], processing power increases faster then consumer Internet bandwidth. The product installations are often in locations where Internet bandwidth increases at an even slower rate comparatively. It makes sense to compress the data before transferring over the obvious bottleneck between the sites and central storage: the public Internet.

Another reason for increasing the file size is metadata overhead. Metadata can account for the majority of space used by a small file, as shown by our experiments with ZFS. The one thing all distributed file systems have in common is that metadata limits their scalability. Metadata needs to scale in order for the file system to scale. In some cases there is no difference in the amount of metadata needed to store a 1 KiB and 1 TiB file; offset and length.

Current usage patterns have no requirement of locking or other POSIX semantics of NFS that have a cost. The datasets are immutable. Batch processing is used to analyze datasets and latency-sensitive real time access is not present. The use cases can be covered with HDFS or an object store combined with a database for metadata. MapReduce would also make it possible to process datasets on the hardware they are stored instead of having to fetch them from HDFS for processing. HDFS-RAID and Xorbas make it possible to have similar low levels of overhead for redundancy that parity-based RAID offers.

5.1 Proposed Storage Architecture

Based on the findings in this Thesis we propose the following system architecture changes to be made in the future. All datasets should be stored as a single file per dataset, preferably several GiB in size. A single file dataset should not simply be an archive of concatenated separate files. The design of ROS bags should be used as guidance. Datasets should be compressed as they are initially produced, and all the tools operate on compressed datasets.

The central storage for datasets should reside in scalable distributed storage. Hadoop and HDFS would be the preferred option, as the combination also enables distributed computations. If MapReduce or HDFS are deemed unsuitable, another distributed file system or object store can be substituted. Dataset metadata should be stored in a database to enable efficient queries.

Vcache should store its index in a database. A better cache invalidation policy should be implemented using the database. Objects stored in Vcache should be moved to an object store or distributed file system. The changes

to Vcache can be implemented in stages.

The proposed changes do limit the ability to use any tools that work in a UNIX environment. Changes would be required not only in the storage architecture, but software producing datasets and processing them. Staying with the current architecture on the other hand limits performance and scalability.

Bibliography

[1] Aghayev, A., and Desnoyers, P. Skylight - A Window on Shingled Disk Operation. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies, FAST 2015, Santa Clara, CA, USA, February 16-19, 2015* (2015), J. Schindler and E. Zadok, Eds., USENIX Association, pp. 135–149.

[2] Amer, A., Holliday, J., Long, D., Miller, E., Paris, J., and Schwarz, T. Data Management and Layout for Shingled Magnetic Recording. *IEEE Transactions on Magnetics 47*, 10 (Oct. 2011), 3691–3697.

[3] Apache HBase Reference Guide. Webpage, 2015. `https://hbase.apache.org/book.html`. Accessed 2015-03-06.

[4] Bairavasundaram, L. N., Goodson, G. R., Schroeder, B., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. An Analysis of Data Corruption in the Storage Stack. In *6th USENIX Conference on File and Storage Technologies, FAST 2008, February 26-29, 2008, San Jose, CA, USA* (2008), M. Baker and E. Riedel, Eds., USENIX, pp. 223–238.

[5] Barratt, C. BackupcPC. Webpage. `http://backuppc.sourceforge.net/`, Accessed 2015-03-02.

[6] Beaver, D., Kumar, S., Li, H. C., Sobel, J., Vajgel, P., et al. Finding a Needle in Haystack: Facebook's Photo Storage. In *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings* (2010), R. H. Arpaci-Dusseau and B. Chen, Eds., USENIX Association, pp. 47–60.

[7] Bonwick, J. The Slab Allocator: An Object-Caching Kernel Memory Allocator. In *USENIX Summer 1994 Technical Conference, Boston,*

Massachusetts, USA, June 6-10, 1994, Conference Proceeding (1994), USENIX Association, pp. 87–98.

[8] Bonwick, J., Ahrens, M., Henson, V., Maybee, M., and Shellenbaum, M. The Zettabyte File System. 2003.

[9] Borthakur, D. HDFS Architecture Guide. Webpage, 2014. `https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html`. Accessed 2015-01-18.

[10] Callaghan, B., Pawlowski, B., and Staubach, P. NFS Version 3 Protocol Specification. RFC 1813 (Informational), June 1995.

[11] Ceph Filesystem. Webpage. `http://ceph.com/docs/master/cephfs/`. Accessed 2015-02-02.

[12] Collet, Y. LZ4: Extremely Fast Compression algorithm. Webpage, 2011. `https://code.google.com/p/lz4/`. Accessed 2015-03-06.

[13] Dean, J., and Ghemawat, S. MapReduce: Simplified Data Processing on Large Clusters. In *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004* (2004), E. A. Brewer and P. Chen, Eds., USENIX Association, pp. 137–150.

[14] Dees, B. Native command queuing - advanced performance in desktop storage. *IEEE Potentials 24*, 4 (Oct. 2005), 4–7.

[15] Elerath, J. G. Hard Disk Drives: The Good, the Bad and the Ugly. *Commun. ACM 52*, 6 (2009), 38–45.

[16] Gallager, R. G. Low-Density Parity-Check Codes. *IRE Transactions on Information Theory 8*, 1 (Jan. 1962), 21–28.

[17] Ghemawat, S., Gobioff, H., and Leung, S. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003* (2003), M. L. Scott and L. L. Peterson, Eds., ACM, pp. 29–43.

[18] Google Snappy: A Fast Compressor/Decompressor. Webpage, 2011. `https://code.google.com/p/snappy/`. Accessed 2014-10-27.

[19] Hamming, R. W. Error Detecting and Error Correcting Codes. *Bell System Technical Journal 29*, 2 (1950), 147–160.

[20] Hancock, S. M. *Tru64 UNIX File System Administration Handbook*. Digital Press, 2001.

[21] Hasenstein, M. The Logical Volume Manager (LVM), 2001.

[22] Heichler, J. Introduction to BeeGFS, Nov. 2014.

[23] Henson, V., Ahrens, M., and Bonwick, J. Automatic Performance Tuning in the Zettabyte File System. *File and Storage Technologies (FAST), work in progress report* (2003).

[24] Hickmann, B., and Shook, K. ZFS and RAID-Z: The Über-FS?, 2007.

[25] Hitz, D., Lau, J., and Malcolm, M. A. File System Design for an NFS File Server Appliance. In *USENIX Winter 1994 Technical Conference, San Francisco, California, January 17-21, 1994, Conference Proceedings* (1994), USENIX Association, pp. 235–246.

[26] Huang, C., Simitci, H., Xu, Y., Ogus, A., Calder, B., Gopalan, P., Li, J., and Yekhanin, S. Erasure Coding in Windows Azure Storage. In *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012* (2012), G. Heiser and W. C. Hsieh, Eds., USENIX Association, pp. 15–26.

[27] illumos. Webpage. `http://wiki.illumos.org/display/illumos/illumos+Home`. Accessed 2015-03-06.

[28] Intel. *Lustre Software Release 2.x: Operations Manual*, Jan. 2015.

[29] Kraemer, F. What's New with GPFS v3.5. Presentation Slides, Apr. 2013.

[30] Krevat, E., Tucek, J., and Ganger, G. R. Disks Are Like Snowflakes: No Two Are Alike. In *13th Workshop on Hot Topics in Operating Systems, HotOS XIII, Napa, California, USA, May 9-11, 2011* (2011), M. Welsh, Ed., USENIX Association.

[31] McKusick, K., and Quinlan, S. GFS: Evolution on Fast-forward. *Commun. ACM 53*, 3 (Mar. 2010), 42–49.

[32] Mushran, S. *OCFS2: A Cluster File System for Linux*. Oracle, July 2008.

[33] Nielsen, J. Nielsen's Law of Internet Bandwidth. Webpage, 2014. `http://www.nngroup.com/articles/law-of-bandwidth/`. Originally published in 1998, but updated with 2014 data. Accessed 2015-01-18.

[34] Nowicki, B. NFS: Network File System Protocol specification. RFC 1094 (Informational), Mar. 1989.

[35] Patterson, D. A. Latency lags bandwith. In *23rd International Conference on Computer Design (ICCD 2005), 2-5 October 2005, San Jose, CA, USA* (2005), IEEE Computer Society, pp. 3–6.

[36] Patterson, D. A., Gibson, G. A., and Katz, R. H. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, June 1-3, 1988.* (1988), H. Boral and P. Larson, Eds., ACM Press, pp. 109–116.

[37] Pawlowski, B., Shepler, S., Beame, C., Callaghan, B., Eisler, M., Noveck, D., Robinson, D., and Thurlow, R. The NFS version 4 protocol. In *Proceedings of the 2nd International System Administration and Networking Conference (SANE 2000)* (2000), vol. 2, p. 50.

[38] Prakash, V., Wen, Y., and Shi, W. Tape Cloud: Scalable and Cost Efficient Big Data Infrastructure for Cloud Computing. In *2013 IEEE Sixth International Conference on Cloud Computing, Santa Clara, CA, USA, June 28 - July 3, 2013* (June 2013), IEEE, pp. 541–548.

[39] Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., and Ng, A. Y. ROS: an Open-Source Robot Operating System. In *ICRA workshop on open source software* (2009), vol. 3, p. 5.

[40] Gluster File System. Webpage. `http://www.gluster.org`. Accessed 2015-02-02.

[41] Red Hat Global File System 2. Webpage, 2014. `https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/7/html/Global_File_System_2/`. Accessed 2015-02-02.

[42] Reed, I. S., and Solomon, G. Polynomial Codes over Certain Finite Fields. *Journal of the Society for Industrial & Applied Mathematics 8*, 2 (1960), 300–304.

[43] Rodeh, O., Bacik, J., and Mason, C. BTRFS: The Linux B-Tree Filesystem. *Trans. Storage 9*, 3 (Aug. 2013), 9:1–9:32.

[44] Rosenblum, M., and Ousterhout, J. K. The Design and Implementation of a Log-structured File System. *ACM Trans. Comput. Syst. 10*, 1 (Feb. 1992), 26–52.

[45] Sathiamoorthy, M., Asteris, M., Papailiopoulos, D., Dimakis, A. G., Vadali, R., Chen, S., and Borthakur, D. XORing Elephants: Novel Erasure Codes for Big Data. In *Proceedings of the VLDB Endowment* (Mar. 2013), vol. 6, VLDB Endowment, pp. 325–336.

[46] Schalnat, G. E., Dilger, A., Bowler, J., Randers-Pehrson, G., et al. libpng - The Official Portable Network Graphics Reference Library. Webpage. `http://libpng.org/pub/png/libpng.html`. Accessed 2015-03-02.

[47] Schmuck, F. B., and Haskin, R. L. GPFS: A Shared-Disk File System for Large Computing Clusters. In *FAST* (2002), vol. 2, p. 19.

[48] Seagate Technology LLC. *Barracuda® XT Product Manual*, Sept. 2010. Revision D.

[49] Seagate Technology LLC. *Constellation® ES Serial ATA Product Manual*, Feb. 2012. Revision D.

[50] Seagate Technology LLC. *Constellation® CS Serial ATA Product Manual*, May 2013. Revision D.

[51] Shepler, S., Callaghan, B., Robinson, D., Thurlow, R., Beame, C., Eisler, M., and Noveck, D. NFS version 4 Protocol. RFC 3010 (Proposed Standard), Dec. 2000. Obsoleted by RFC 3530.

[52] Shepler, S., Callaghan, B., Robinson, D., Thurlow, R., Beame, C., Eisler, M., and Noveck, D. Network File System (NFS) version 4 Protocol. RFC 3530 (Proposed Standard), Apr. 2003.

[53] Shvachko, K. V. HDFS Scalability: The limits to growth. *login 35*, 2 (2010), 6–16.

[54] Subramanian, M., Lloyd, W., Roy, S., Hill, C., Lin, E., Liu, W., Pan, S., Shankar, S., Viswanathan, S., Tang, L., and Kumar, S. f4: Facebook's Warm BLOB Storage System. In *11th*

USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014. (2014), J. Flinn and H. Levy, Eds., USENIX Association, pp. 383–398.

[55] Sun Microsystems, Inc. *ZFS On-Disk Specification*, 2006. Draft.

[56] Sweeney, A., Doucette, D., Hu, W., Anderson, C., Nishimoto, M., and Peck, G. Scalability in the XFS File System. In *Proceedings of the USENIX Annual Technical Conference, San Diego, California, USA, January 22-26, 1996* (1996), USENIX Association, pp. 1–14.

[57] Tanenbaum, A. S. *Modern Operating Systems*, 3rd ed. Prentice Hall Press, Upper Saddle River, NJ, USA, 2007.

[58] Tridgell, A., and Mackerras, P. The rsync algorithm. Tech. rep., Australian National University, June 1996.

[59] Vallamsetty, U. ZFS Write Performance (Impact of fragmentation). Webpage, Feb. 2013. `http://blog.delphix.com/uday/2013/02/19/78/`. Accessed 2015-03-06.

[60] Wang, W., and Hairong, K. Saving Capacity with HDFS RAID. Webpage, June 2014. `https://code.facebook.com/posts/536638663113101/saving-capacity-with-hdfs-raid/`. Accessed 2014-10-27.

[61] Weil, S. A., Brandt, S. A., Miller, E. L., Long, D. D. E., and Maltzahn, C. Ceph: A Scalable, High-Performance Distributed File System. In *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA* (2006), B. N. Bershad and J. C. Mogul, Eds., USENIX Association, pp. 307–320.

[62] Zou, H., Yu, Y., Tang, W., and Chen, H. M. Improving I/O Performance with Adaptive Data Compression for Big Data Applications. In *2014 IEEE International Parallel & Distributed Processing Symposium Workshops, Phoenix, AZ, USA, May 19-23, 2014* (2014), IEEE, pp. 1228–1237.

Appendix A

Block Size Effect on Throughput

```
# vim: set si ai et sw=4 sts=4 ts=4 ft=python:
import ctypes
import os
import random
import sys
import time

LIBC = ctypes.CDLL('libc.so.6')
SECTOR = 512

def drop_caches():
    LIBC.sync()
    with open('/proc/sys/vm/drop_caches', 'w') as drop:
        drop.write('3\n')  # drop pagecache, dentries and inodes

def main():
    drive = open('/dev/sda', 'rb+')
    drive.seek(0, os.SEEK_END)
    length = drive.tell()

    def randread(block, count):
        read = 0
        maxsector = int((length-block)/SECTOR)
        for i in range(count):
            drive.seek(random.randint(0, maxsector)*SECTOR)
            read += len(drive.read(block))
        return read

    for i in range(30):
```

```
        block = 2**i
        drop_caches()
        start = time.time()
        read = randread(block, 1000)
        delta = time.time() - start
        print 'block:', block, 'secs:', delta, 'read:', read

if __name__ == '__main__':
    main()
```

Appendix B

ZFS Metadata Overhead

```
#!/bin/sh -e
# vim: set si ai et ts=4 sts=4 sw=4 ft=sh:
for size in 0 1 128 1024; do  # kB
    for count in 10 100 1000 10000 100000; do
        # create pristine file system
        zfs create tank/test
        for i in $(seq -w 0 $(($count-1))); do
            dd if=/dev/urandom of=/tank/test/$i \
                bs=1k count=$size > /dev/null 2>&1
        done
        # force changes to disk by exporting and importing
        zpool export tank
        zpool import tank
        echo "$size $count"
        zfs list -p tank/test
        # cleanup
        zfs destroy tank/test
    done
done
```

Appendix C

NFS Latency Overhead

```
# A collection of commands used to measure NFS latency

# Used to drop the caches
dropcache() {
    echo 3 > /proc/sys/vm/drop_caches
}

# Use external time(1) to be able to redirect stderr
alias time='command time'

# The path where the files are located
DIR="$1"

# Measure time taken to `find -ls` the directory
time -f '%e' find "$DIR" -ls >/dev/null 2>find.time

# Measure time taken to `find -exec file` the directory
time -f '%e' find "$DIR" -exec file '{}' \; >/dev/null 2>file.time

# List full path of files with selected size to cat.files
find "$DIR" -type f -size +900k -size -1000k >cat.files

# Count the number of bytes read from files listed in cat.files
time -f '%e' xargs cat 2>cat.time <cat.files|wc -c

# Split file contents into 100MiB files
xargs cat <cat.files |split -b100M
```