LoneStar: Design and Evaluation of an Energy-Efficient, Disk-Based Archival Storage System

Dissertation
zur Erlangung des Grades
„Doktor der Naturwissenschaften“
am Fachbereich Physik, Mathematik und
Informatik der Johannes Gutenberg-Universität
in Mainz

vorgelegt von
Matthias Grawinkel

Mainz, 2015

Abstract

LoneStar: Design and Evaluation of an Energy-Efficient,
Disk-Based Archival Storage System
by
Matthias Grawinkel

Tape has been the predominant storage target for digital archives, but with workloads becoming more active and storage media like hard disk drives catching up on costs-per-byte, the archival storage stack needs to be reinvestigated. Reliability, integrity, and durability are the main goals of digital preservation, but the role of performance increases with active archives that provide near-online access to all stored data. A tape-only solution cannot deliver the required parallelism, latencies, and throughput, which leads to disk-based caches as compensation.

In this thesis, we explore the challenges and opportunities of building a hard disk drive based storage system that aims to be highly reliable and energy efficient and is suited for active archives as well as cold storage environments. First, we analyze the storage landscape and access behavior of a large digital archive to present a use case for the proposed architecture. Then, we develop mechanisms to improve the reliability of a single disk drive and propose and evaluate a new energy efficient two-dimensional RAID scheme that is optimized for “write once, read sometimes” workloads. Finally, we present journaling and caching schemes that support the underlying goals and evaluate the RAID scheme in a file system environment.

Zusammenfassung

LoneStar: Design and Evaluation of an Energy-Efficient,
Disk-Based Archival Storage System
von
Matthias Grawinkel

Bandlaufwerke waren bisher die vorherrschende Technologie, um die anfallenden Datenmengen in Archivsystemen zu speichern. Mit Zugriffsmustern, die immer aktiver werden, und Speichermedien wie Festplatten die kostenmäßig aufholen, muss die Architektur vor Speichersystemen zur Archivierung neu überdacht werden. Zuverlässigkeit, Integrität und Haltbarkeit sind die Haupteigenschaften der digitalen Archivierung. Allerdings nimmt auch die Zugriffsgeschwindigkeit einen erhöhten Stellenwert ein, wenn aktive Archive ihre gesamten Inhalte für den direkten Zugriff bereitstellen. Ein bandbasiertes System kann die hierfür benötigte Parallelität, Latenz und Durchsatz nicht liefern, was in der Regel durch festplattenbasierte Systeme als Zwischenspeicher kompensiert wird.

In dieser Arbeit untersuchen wir die Herausforderungen und Möglichkeiten ein festplattenbasiertes Speichersystem zu entwickeln, das auf eine hohe Zuverlässigkeit und Energieeffizienz zielt und das sich sowohl für aktive als auch für kalte Archivumgebungen eignet. Zuerst analysieren wir die Speichersysteme und Zugriffsmuster eines großen digitalen Archivs und präsentieren damit ein mögliches Einsatzgebiet für unsere Architektur. Daraufhin stellen wir Mechanismen vor um die Zuverlässigkeit einer einzelnen Festplatte zu verbessern und präsentieren sowie evaluieren einen neuen, energieeffizienten, zweidimensionalen RAID Ansatz der für „Schreibe ein Mal, lese mehrfach“ Zugriffe optimiert ist. Letztlich stellen wir Protokollierungs- und Zwischenspeichermechanismen vor, die die zugrundeliegenden Ziele unterstützen und evaluieren das RAID System in einer Dateisystemumgebung.

Personal Publications

[CGNB12] G. Congiu, M. Grawinkel, S. Narasimhamurthy, and A. Brinkmann. One Phase Commit: A Low Overhead Atomic Commitment Protocol for Scalable Metadata Services. In *Proc. of the 4th Workshop on Interfaces and Abstractions for Scientific Data Storage (IASDS)*, 2012.

[CGP+14] G. Congiu, M. Grawinkel, F. Padua, J. Morse, T. Süß, and A. Brinkmann. Optimizing Scientific File I/O Patterns using Advice Based Knowledge. In *Poster at the IEEE International Conference on Cluster Computing (CLUSTER)*, 2014.

[DG07] M. Ditze and M. Grawinkel. Fuzzy Logic Based Admission Control for Multimedia Streams in the UPnP QoS Architecture. In *Proc. of the 21st International Conference on Advanced Information Networking and Applications Workshops (AINAW)*, 2007.

[GBSB14] M. Grawinkel, G. Best, M. Splietker, and A. Brinkmann. LoneStar Stack: Architecture of a Disk-Based Archival System. In *Proc. of the 9th IEEE International Conference on Networking, Architecture, and Storage (NAS)*, 2014. ©2014 IEEE. Reprinted, with permission, from M. Grawinkel and G. Best and M. Splietker and A. Brinkmann, LoneStar Stack: Architecture of a Disk-Based Archival System, August 2014.

[GMPB11] M. Grawinkel, H. Dömer M. Pargmann, and A. Brinkmann. LoneStar: An Energy-Aware Disk Based Long-Term Archival Storage System. In *Proc. of the 17th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, 2011.

[GNB15] M. Grawinkel, L. Nagel, and A. Brinkmann. LoneStar RAID: Massive Array of Offline Disks for Archival Systems. *Submitted to ACM Transactions on Storage (TOS)*, 2015.

[GNM+15] M. Grawinkel, L. Nagel, M. Mäsker, F. Padua, A. Brinkmann, and L. Sorth. Analysis of the ECMWF Storage Landscape. *Proc. of the 13th USENIX Conference on File and Storage Technologies (FAST)*, 2015.

[GSB+11] M. Grawinkel, T. Schäfer, A. Brinkmann, J. Hagemeyer, and M. Porrmann. Evaluation of Applied Intra-Disk Redundancy Schemes to Improve Single Disk Reliability. In *Proc. of the 19th IEEE International Symposium on Modeling, Analysis, and Simulation (MASCOTS)*, 2011. ©2011 IEEE. Reprinted, with permission, from M. Grawinkel and T. Schäfer and A. Brinkmann and J. Hagemeyer and M. Porrmann, Evaluation of Applied Intra-Disk Redundancy Schemes to Improve Single Disk Reliability, July 2011.

[GSB+12] M. Grawinkel, T. Süß, G. Best, I. Popov, and A. Brinkmann. Towards Dynamic Scripted pNFS Layouts. In *Proc. of the 7th Petascale Data Storage Workshop (PDSW)*, 2012.

Acknowledgements

[The acknowledgments have been removed from the online version of the thesis due to regulations of the Johannes Gutenberg University, Mainz]

Contents

1 General Introduction **1**
1.1 Storage Media . . . 3
1.2 Reliability, Checksums, and Error Correcting Codes . . . 5
1.3 Many Disks and Storage Systems . . . 6
1.4 Organization & Contributions of this Thesis . . . 9

2 Analysis of the ECMWF Storage Landscape **12**
2.1 Introduction . . . 13
2.2 Related Work . . . 14
2.3 Background . . . 15
2.3.1 Available Log Files . . . 16
2.4 ECFS User Archive . . . 17
2.4.1 Metadata Snapshot Analysis . . . 17
2.4.2 Workload Characterization . . . 20
2.4.3 User Session Analysis . . . 21
2.5 MARS Database . . . 24
2.5.1 Metadata Snapshot Analysis . . . 25
2.5.2 Workload Characterization . . . 26
2.6 Tape Mount Logs . . . 29
2.6.1 Tape Prefetching . . . 32
2.7 Cache Simulation . . . 33
2.8 Discussion & Conclusion . . . 37
2.9 Closing Remarks . . . 38

3 Evaluation of Applied Intra-Disk Redundancy Schemes to Improve Single Disk Reliability **39**
3.1 Introduction . . . 40
3.2 Related Work . . . 41
3.3 Challenges of applying IDR codes . . . 42
3.4 Parity Group Scrambling . . . 43

3.5 Evaluation . . . 44
3.5.1 Power Consumption . . . 46
3.5.2 iSCSI Access Patterns . . . 47
3.5.3 Intra-Disk Redundancy Codes . . . 48
3.5.4 Full I/O stack . . . 49
3.5.5 Reliability comparison . . . 53
3.5.6 Check on Read Overhead . . . 55
3.6 Conclusion and Discussion . . . 55

4 LoneStar RAID: Massive Array of Offline Disks for Archival Systems 57
4.1 Introduction . . . 58
4.2 Related Work . . . 59
4.3 LoneStar RAID . . . 61
4.3.1 Intra-Disk Redundancy . . . 62
4.3.2 Intertwined RAID Scheme . . . 63
4.3.3 Journaling & Promises . . . 65
4.3.4 Caching . . . 66
4.3.5 Write Modes & Semantics . . . 68
4.3.6 Recovery & Maintenance . . . 69
4.3.7 Performance . . . 71
4.4 Evaluation . . . 73
4.4.1 Reliability Model . . . 73
4.4.2 Mean Time To Dataloss (MTTDL) Analysis . . . 75
4.4.3 Suitable Configurations . . . 76
4.4.4 Prototype & Simulation Environment . . . 77
4.4.5 Simulated Disk Backend . . . 79
4.4.6 Simulation Results . . . 81
4.5 Discussion & Conclusion . . . 87

5 LoneStar Stack: Architecture of a Disk-Based Archival System 91
5.1 Introduction . . . 92
5.2 Related Work . . . 93
5.3 LonesStar Storage Stack . . . 94
5.3.1 LoneStar RAID . . . 95
5.3.2 LoneStar FS . . . 96
5.3.3 Reading files . . . 98
5.3.4 Writing files . . . 99
5.3.5 File Deletion . . . 100
5.3.6 Placement Engine & Energy Management . . . 100
5.3.7 Management & RPC . . . 101
5.3.8 Consistency, Reliability, and Recovery . . . 102

5.4 Evaluation . . . 103
5.4.1 Implementation . . . 103
5.4.2 Evaluation Hardware . . . 104
5.4.3 Evaluation Design . . . 104
5.4.4 LoneStar FS Overhead . . . 105
5.4.5 Read Performance . . . 106
5.4.6 Write Performance . . . 107
5.5 Conclusion & Future Work . . . 108

6 Conclusion **110**
6.1 Future Work . . . 112

Glossary **117**

Bibliography **118**

List of Figures **131**

List of Tables **134**

1 | General Introduction

The development of today's civilization was only possible through cultural techniques that preserve knowledge and make it accessible for others and future generations. Symbols or pictographs carved into stones, handwritten or typeset writings, or zeros and ones encoded to magnetized grains of a hard disk drive platter are just some examples how human beings recorded and preserved information. The culture of writing, the invention of the letterpress, and the immediate and unlimited reproducibility of digital information present an ever accelerating path of information processing. Before the computerization and digitization of information processing, the speed of interpreting recorded information was limited by the capabilities of human brains. Today, gigantic collections of digital information exist, that can be publicly accessed, searched, and processed within seconds.

The actual value of any piece of information is not always immediately clear. Any datum which itself possibly was irrelevant at the time of its creation, can become valuable at a later point in time, or relevant in a broader context like a measurement point in a time series. With today's ever decreasing costs for digital storage, the global trend is to store as much information as possible. Also, it can be cheaper to just store and keep information than to decide on its value and maybe later delete it.

Many different digital archives exist, which continuously grow and are designed to keep stuff forever. The non-profit organization *Internet Archive* [JK09] is just one example. Driven by their mission statement "universal access to all knowledge", they have aggregated about 10 petabytes of websites, music, moving images and books as of 2012. All archived information is publicly accessible. The Church of Jesus Christ of Latter-day Saints, as another example, has a scriptural mandate to keep records of its proceedings and is doing so since 1830 [Wri12]. Their *FamilySearch* project alone, which archives 13.1 billion images of historical and vital records, approaches 100 petabytes within this decade. Another large-scale domain-specific archive was analyzed within this thesis. The European Centre for Medium-Range Weather Forecasts (ECMWF) operates since 1975 and has aggregated the world's largest collection of meteorological data. Internally, this archive is used by their supercomputers to calculate models for weather forecasts. In addition to that, the archive can be accessed by all ECMWF's member states and external paying customers. Today, they provide a total storage capacity of 100 petabytes and face a compound annual growth of about 50%.

Running such long-term services requires a large technological stack, experienced personnel, and a long-term commitment. Natural disasters, human errors, and economical or technical issues are only some threats to digital preservation, which have been identified in multiple publications [BKM05, BSR+06, Ros10, RRM+12]. In the following, we will focus on the technological aspects of long-term digital preservation, as well as the energy consumption as it is an important factor of the running storage costs.

From a technical perspective, the reliability, consistency, and integrity of stored data are the most important requirements for a digital archive. An archived document is expected to be available and unchanged for as long it should be stored. Nevertheless, data can be modified or get lost due to human errors or failing soft- or hardware. If multiple copies of files exist on different storage systems and locations, the data can be restored from another source. If data from a single-source archive like the ECWMF is lost, it cannot be restored from another copy. While the result of a computational model can be recalculated, a historic measurement point will be lost forever. Depending on the importance of the data and the option to recover the information, the architecture to preserve data will change. “Lots Of Copies Keep Stuff Safe”, for example, is the name and idea behind the software *LOCKSS* - a distributed storage system designed for libraries to securely store and preserve data [MRG+05]. The availability of multiple copies of data allows to recover the loss of some copies and the integrity of data can be checked by consensus. If one or multiple copies differ, they will be replaced by the majority of the other copies that are the same. So, the more copies of a file exist, the more robust it will become against tampering, censorship, and accidental loss of data. This system works great for storage systems with medium sized data volumes, but maintaining many copies of a full multi-petabyte sized archive is not yet feasible for economical reasons.

There is no archetypical archive as its usage and growth depends on many factors like access patterns of new, updated and read data, requirements on performance, latency, availability, reliability and available money to fulfill these goals. The architecture of a cold archive that is mostly written and accessed rarely might focus on a tape only strategy. An active archive on the other hand, where all stored data is subject to be read at any point in time will focus on a more responsive storage stack. Here, a compromise of required reliability, performance, latency, and parallelism needs to match economical boundaries. An archive is built to survive many years and typically grows over time. Therefore, its physical representation continuously evolves and can consist of multiple generations of storage media and various storage systems.

This thesis develops the LoneStar architecture - an archival system based on hard disk drives to store data and non-volatile ram (NVRAM) to improve the overall system efficiency. The following sections provide a general introduction and discussion of challenges and technologies that influenced the design of LoneStar. First, we will outline the characteristics of diverse storage media and justify our focus on hard disk drives and NVRAM. Then, we provide an overview of threats to data storage induced by media errors and how to cope with them. Finally, we discuss how many disk drives can be used to improve the

reliability and performance of storage systems and how their usage influences the overall energy consumption of the storage stack.

1.1 Storage Media

Digital information is represented in bits - the single smallest unit that is either zero or one. A bit stream is an arbitrary sequence of bits and an encoding defines how to interpret the inherent information. A sequence like 10010111, can, for example, be interpreted as 8 booleans, as an integer, as characters of a string, or as a part of an audio waveform. The main characteristic of a storage system is that an exact copy of stored data can be retrieved again. Not even a single bit of stream must be flipped. While a minimal error in the bit stream of an image or video might not even be visible, a binary flag going from 'no' to 'yes' can have severe consequences.

The bits of digital information are persisted to physical media. Over the years, many different technologies were introduced that differ in access performance, density, price and durability. A noteworthy extreme was the *Voyager Golden Record*, which was built by a team of scientists around Carl Sagan. It is a phonograph record, which contains selected sounds and images of life and culture on earth on the A-side and an instruction how to read it is engraved as pictographs on the B-side. The disk was designed to endure 500 million years and in 1977 it was sent to space as a "bottle into the cosmic ocean" to at least provide record of existence of human kind [Sag78]. In contrast, a modern hard disk drive provides a density and capacity that is multiple orders of magnitudes higher while also being much cheaper. On the other hand, its lifetime is expected to be less than a decade. Even worse, the disk drive's electronic components might crash before that and all stored information would be lost.

In this thesis, we will focus on tape, hard disk drives, and flash as they are most commonly found in data centers and computers. Their very different performance, durability and price characteristics are relevant for different usage scenarios in information processing. From an economical point, tape is very relevant as it can preserve huge amounts of data for a long time for a considerable cheap price. The actual storage media is decoupled from the drives that are required to read or write the data. A single tape drive can be shared by many tape cartridges that are stored in large libraries. A robot then moves the cartridges between the library and the available drives. On the other hand, tape is impractical if the stored data is read and written in regular intervals and from multiple readers and writers in parallel. To read a single bit, a tape robot needs to fetch and mount the cartridge into a drive, which then winds and reads the tape. This process takes multiple seconds and, if more requests arrive in parallel than drives are available, access latencies can grow to many minutes.

The general concept of a hard disk drive (HDD) was introduced by IBM in 1956[1] and has seen a constant stream of innovation and advancement of performance, capacity,

[1] http://www-03.ibm.com/ibm/history/exhibits/storage/storage_350.html

reliability, and energy-efficiency since then. In the simplest form, a single disk drive enclosure contains a single platter with a single actuator arm. The platter is coated with a magnetic material and the actuator arm moves a magnetic head to read and write data from and to the platter's surface. Conceptionally, a disk drive works similar to a phonograph. Data can only be accessed, if the drive is powered up and the disk is spinning at a constant speed. To read a phonograph record, a needle follows the spiral groove that usually starts near the periphery and ends near the center of the disc. Unlike that, the magnetic head of an HDD floats above the platter without physically touching it. Also, the disk contains many concentric tracks, and without deliberately moving the head, only a single track can be accessed. A track consists of sectors that store 512 bytes (4096 bytes on newer disks). The bit-density within a disk does not change, so that each sector requires the same length. A disk is partitioned into multiple zones that are characterized by having the same number of sectors per track. Accordingly, the read and write performance of a disk is higher at the outer tracks of the disk, as more sectors pass the magnetic head per round than for the inner tracks of the disk.

In today's high capacity disk drives the spindle has multiple platters and an arm assembly holds an arm with a magnetic head for each side of a platter [RW94]. When a disk is started, the spindle requires a lot of power until the disk platters rotate with - depending on the model - 4,200 up to 15,000 rounds per minute. The time between powering a disk until it is fully operable depends on the form-factor, number and weight of platters, and the target rotation speed. For a current off-the-shelf 3,5" consumer hard disk drive a spin-up takes around 10 seconds, while a 2,5" drive is operable in less than 5 seconds. To access a track on the disk, the arm moves to the target track, the head is positioned, and when the target position of the rotating platter arrives, it is accessed. Disk drives play an important role in today's storage systems because of their price, capacity and sequential throughput capabilities. Unfortunately, for random accesses, they suffer from physical constraints. Even the fastest drives require 4 ms as an average access time, which results in a theoretical maximum of 250 random operations per second.

To overcome the physical limitations of disk drives, many different possibilities have been explored. Every disk contains a small cache to buffer and aggregate writes to plan and optimize required disk head moves. The best performance for a disk drive can be achieved if the physical conditions and capabilities can be exploited as far as possible. The fast file system [MJLF84] for example was designed to store related data and metadata blocks close to each other to reduce disk head movements and therefore improve performance. A trend that was followed by most of the following file systems designed for HDDs. Gim et al. have shown that the performance of disk accesses can be significantly improved for known hard disk drive characterizations [GWC+08]. The effects of track alignments have also been investigated in other contexts [SGLG02, QMW08, LMB10].

Hard disk drives require a constant energy supply to keep the disks spinning. Over time, different approaches were investigated how the overall energy consumption can be reduced. Disks use the least energy, when they are not spinning. On the other hand,

a spin-up causes additional latency, wear-out, and increased energy consumption during startup. If it is worthwhile to spin down a disk heavily depends on the workloads and their requirements on latency. Mobile computing was the initial driver to reduce disk power consumption. Especially the trade-off between idle time until spin-down and induced latencies for wake-ups were investigated [DKM94, LKHA94, HLS96]. Multiple works investigated the creation of disk drive energy models to foster the development of power saving techniques and disk drive simulators [Gre94, ZSG+03, HSRJ08, HS09]. Others investigated multi-speed disks and acoustic modes to use less energy [CPB03, CGK+10]. Bisson et al. added an NVRAM-based cache to support the spin-down of a single disk [BBL06]. They showed that a hard disk's power consumption could be reduced by up to 90%. The case where disk-idle times are too small to be exploited was investigated by Gurumurth et al. [GAKF03] who proposed to dynamically adjust the RPM of disks. Running a disk at a slower speed saves energy without incurring the latency and power overhead of a full spin-down. Also, requests that don't require a high performance can be served by the disk running in the slow state. If full performance is required, the platters are accelerated again.

In the last years, flash-based solid-state drives (SSD) gained a lot of importance and became the predominant storage medium in today's consumer electronics and compute environments that require high throughput and parallelism. An SSD has no mechanical parts and uses non-volatile flash memory chips as a storage medium. On a costs-per-byte scale, flash is still more expensive than hard disk drives. On the other hand, it outperforms disk drives by orders of magnitudes in terms of throughput and operations per second. Also, it provides better energy efficiency, as no mechanical parts need to be spun up and kept spinning. Many of today's storage systems use hybrid solutions that combine the performance and capacity characteristics of flash-based media and the hard disk drives. Such a hybrid can be a SSHD that adds a flash-based cache to a hard disk drive or software or hardware cache in a multi-layer storage architecture.

1.2 Reliability, Checksums, and Error Correcting Codes

Magnetic disk drives are still the most commonly applied devices to store data, and they are used in many environments where data losses cannot be tolerated, like in file servers, archives, or backup systems. Next to complete disk failures, additional error types like sectors that can no longer be read by the disk, called latent sector errors (LSE), or reads that deliver wrong results, called block corruptions, have to be considered.

The probability of latent sector errors and corrupt blocks has not increased per access over the last years. Nevertheless, the increasing bit and track densities of today's hard disk drives and their resulting exponentially growing capacities have led to an increase in the amount of stored data that makes this small probability very relevant. Studies on disk failures revealed that, next to full disk failures, LSEs and corrupt blocks are a big threat to data reliability [BADAD+08]. They are especially difficult to handle because they are only detected when corresponding sectors are read.

To verify the integrity of a bit stream, a checksum function can be applied to an arbitrary block of digital data. Typically, these checksums are calculated by cryptographic hash function like MD5 or SHA-3 that result in another bit stream. Depending on the chosen algorithm this checksum can consist of up to 512 bits. These bits can either be stored next to the data or at another location. Whenever the data is read, the checksum is recalculated and compared to the previously stored version. If they differ, the bit stream is corrupt.

Though checksums can prove the integrity of data, they cannot be used for recovery. Originally, erasure codes were developed as a forward error correction to recover from errors during data transmission. By adding k redundancy blocks to a bit stream of n blocks, any n of the $n + k$ blocks can be used to calculate the original bit stream. This mechanism can be applied to storage media to create an intra-disk redundancy (IDR). If a data block cannot be read or the checksum fails, the redundancy blocks can be used to restore the erroneous data. Many different codes have been created or adapted for IDR and were analyzed for their reliability and performance [RS60, BBBM95, CEG+04, PBA+05, Haf05, WT07, Pla08b, DEH+08, IHHE08, PLS+09, SDG10].

Corrupt blocks and sector errors are only detected, when data is accessed. Every additional error lessens the probability of a successful rebuild. So especially for disk-based archival systems, where data may be written once and never read, the disk's contents should be checked and repaired in regular intervals. This process is called disk scrubbing and its importance was analyzed in multiple publications [SXM+04, IHHE08, GMB10, PSAL10].

1.3 Many Disks and Storage Systems

For many applications the capacity, performance, or reliability of a single disk is not enough. Luckily, all these factors can be improved by adding more disks to the storage stack. On the other hand, these factors heavily influence each other. Multiple disks can either be hosted within a single server, distributed over multiple servers, be part of a storage area network (SAN), or being made available individually by adding a small CPU and a network interface to each disk drive. In the following, we will briefly describe some mechanisms to make use of multiple disks.

To improve the write performance, data blocks can be spread over multiple individual disks. This mechanism also improves the read performance, as multiple disks share the load, but the reliability is heavily decreased as an error on any disk results in data loss. Patterson et al. described this mechanism as RAID0 in 1988 [PGK88]. They also described a RAID1, which mirrors the same data to multiple disks. In the best case this scheme achieves the write performance of a single disk, but data could be read in parallel from multiple disks. Also, any but one of the mirrored disks can fail without loosing data. Patterson et al. also described the RAID4 scheme, which is very relevant in the context of this thesis. For n data disks, one additional parity disk is added to the RAID, that contains

the XORed content of the data disks. The result is a 1 disk failure tolerant (1-DFT) system that can compensate an erroneous or lost disk.

The term RAID6 covers the family of 2-DFT RAID codes like [BBBM95, Pla08b, Pla08a]. Though much more reliable than the original RAID codes proposed by Patterson et al., Leventhal argues that even double parity codes are no longer resilient enough for today's storage systems [Lev09]. Qin et al. state that two-way mirroring provides a sufficient reliability in large storage systems [XMS+03], but recommend a triplication or two-way mirroring combined with RAID for scenarios that require a very high reliability. This level of reliability is achieved by modern distributed file systems like Ceph [WBM+06], which store three replicas of a file by default. Storing and accessing multiple replicas is simple and efficient, but causes a significant storage overhead. Another mechanism to create strong and resilient coding schemes is the use of erasure codes. They have been reinvestigated for performance, overhead, and recoverability [RS60, HSX+12a, SAP+13, PGM13]. A huge difference in applying multiple replicas or erasure coding lies in the updateability of stored data, which becomes visible by examining the update of a single byte of a file. In the replication case, the according byte is updated in every file, which results in a written byte per replica. Given a 12+4 erasure code, the updated byte is written to the according block of the 12 original data blocks and the 4 parity blocks need to be recalculated. In the end, each update results in reading 12 and writing 5 blocks. This shows that erasure coding can provide a good space efficiency and high reliability, but it should only be used in environments, where data is rarely updated.

The tradeoff between multiple replicas and erasure coding is reflected in today's cloud storage architectures that use replication and coding at different stages of a file's lifecycle. First, multiple copies of an object are stored. If the data is not changed or accessed any more, it is erasure encoded and the copies are deleted. The coding schemes used in [HSX+12a, SAP+13] result in a storage overhead of 133%, while providing an even better resiliency than a triplication with a storage overhead of 300%. To provide the resiliency, the encoded blocks of an object need to be stored on different errors domains, like multiple individual hard disk drives. An object encoded with a 12+4 erasure code, for example, should be spread to 16 different disks. To read the object, any 12 of these 16 disks need to be available. While being space efficient and reliable, the system cannot be used efficiently in disk-based environments that shut down disks to save energy.

If the immediate access and high performance of a storage system are not the primary goals, many mechanisms can be applied to improve the reliability or the energy efficiency. First ideas to adapt RAID schemes to workloads was proposed by Wilkes et al. as AutoRAID [WGSS96]. Here, two different RAID schemes are defined on a set of disks. A faster front-end tier with both performance and reliability and a slower tier that aims for reliability. The software-defined system moves data between the tiers to achieve a high performance. Other ideas followed this trend and integrated means to also reduce the energy consumption [LW04, ZCT+05, WOQ+07, GLM+08]. In 2002, Colarelli proposed a massive array of idle disks [CGN02]. If a disk drive is not accessed for some time, it is pow-

ered down to save energy. This shows, that known data access patterns can be exploited to move data between different storage layers.

If data is primarily stored for the sake of preservation and not for access, more extreme measures can be taken to improve the reliability and energy efficiency. One of the more popular disk-based archival systems is Pergamum [SGMV08]. The system was built with enclosures that contain a single disk drive with a small CPU and a network interface. Erasure codes are used to spread data over many individual disks, that each verify their contents with complex hashing schemes. Each disk provides means to check and repair itself and actively powers itself down. This system has the same properties as the aforementioned erasure coded cloud storage systems. To read or write data, multiple disks need to be accessed. The other extreme of many disks per CPU was investigated by Balakrishnan et al. in the Pelican storage system [BBD+14]. The authors proposed an archival storage building block that uses 1,152 archive-grade hard disks powered by two servers. They state that this architecture is right-provisioned for the expected workload of cold storage where access latencies are not the primary goal and accesses can be scheduled beforehand. The hardware was co-designed with the software stack to achieve a target throughput of sustained 40 Gbps, which requires 50 active disks. The disks are organized in power and coding domains to meet hardware induced constraints. For example, the power supplies are under-provisioned and can only power some of the disks at the same time. A careful data placement and scheduling of accesses results in a dense and energy efficient building block that can achieve the target throughput while keeping the initial and running costs of the system low.

Gibson et al. first introduced the use of multi-dimensional parity schemes in 1989 [GHK+89], where each disk belongs to exactly one horizontal and one vertical RAID 4 stripe. Schwarz et al. build on this work and analyze its reliability [SB93]. This RAID scheme can recover all simultaneous 2-disk failures, but can also recover some situations where more than two disks fail at the same time. This approach was improved by Pâris who add a super-parity disk [PSAL12], which improves the system to also be able to recover all 3-disk failures. When all disks of a parity group are fully populated, the super-parity is calculated and written to a dedicated disk. While this scheme improves the overall reliability, it can only be applied if data is mostly written and updated rarely, as the super-parity disk reflects the changes of all other parity disks. Therefore, any write on the RAID results in an update to the super-parity. Pâris et al. also proposed and theoretically evaluated a three-dimensional redundancy code for archival storages [PLL13].

It is important to note that especially for archival systems the strategy how to place objects to storage targets is very important. While systems like the aforementioned dynamic RAID schemes move data between a fast and a reliable or low-energy tier, data in a cold storage tier is not expected to move. Once data is written, it will not change and should not be moved again. Especially for systems where reads and writes involve many disks, like in Pergamum or the multi-dimensional schemes with a super-parity disk, changes to data require many disks to be updated. In the best case, the initial placement decision

for an archived object is final and optimal in respect to future accesses. Previous work optimized data placement schemes in distributed systems to be scalable, reliable and efficient. If the load and accesses are evenly spread over existing resource, on average the best performance can be achieved [BSS00, WBM+06, MEK+11]. For archival workloads where resources shall be powered down, these strategies render the worst case. Here, semantic data placement strategies as proposed by Wildani and Miller present a promising mechanism to map data to storage systems [WM10]. The basic idea is to find clusters of related data and store them close to each other. For example, a set of related documents should be stored to the same set of disks. If one document is accessed, chances are high, that the other documents will be read as well. In the best case, a single disk drive is spun up that stores all documents. In the worst case, the documents are erasure coded and spread to non-overlapping sets of disks that all need to be spun up to read the blocks that make up the documents.

1.4 Organization & Contributions of this Thesis

In this thesis we will focus on the question if hard disk drives can be used as a building block for long-term digital archives. To achieve this, we first analyze a large digital archive to foster the understanding of access patterns of existing systems. Then, we describe the architecture of the disk-based storage system LoneStar that can be used for both, cold as well as active archive environments. Such a system needs to meet multiple requirements.

Once written, a file needs to be stored safely and unaltered until it is either deleted or migrated to the next generation of the storage system. Then, the parallelism and access performance for both, reads and writes has to outperform tape-based solutions. Furthermore, the system needs to provide mechanisms to detect erroneous media and corrupt files, and to recover them. As argued by Qin and Leventhal, such a solution should at least be 3-disk failure tolerant [XMS+03, Lev09].

The costs of a disk-based archive are an important factor, if it should compete with a tape-based solution. Neglecting any housing overhead, in general, a system will consist of a high number of disk drives, enclosures that contain them, servers that operate them, and power supplies that keep all components running. To achieve a target usable capacity for an archive, the number of disk drives and the applied reliability mechanisms define the main architecture. In the best case, a RAID scheme or erasure code is used that provides a low storage overhead, while achieving the target reliability. To reduce the other overhead, the resulting system should be very dense in terms of required space in a rack and number of required servers to operate the archive. Last but not least, the energy costs to run an archive are an important economical factor. Here, a trade-off between performance, availability and responsiveness of the system has to be found. If all data must be immediately available with a high throughput, most of the components are required to be powered up and running. If, on the other hand, data is accessed infrequently, and a short timespan to start components can be tolerated, mechanisms to save energy can be applied. In the best

case, all components are fully shut-off and on access only a minimum number of servers and disks are required to power up.

This thesis is organized as a collection of self-contained, refereed and submitted publications that describe the different details of the LoneStar archival system.

- Chapter 2: *Analysis of the ECMWF Storage Landscape* appeared in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)* [GNM+15].
  In this work we analyze the storage systems and their access patterns of a digital archive from the weather-forecasting domain. The main contribution of the study is the first in depth-analysis of an active archive in the field of storage system research. Within the study, we analyze the access patterns of the systems and the resulting workloads on the tape-based storage backend. We show that disks play an important role in today's large-scale storage archives and are an integral component of the ECMWF's storage stack to deliver the required performance.

- Chapter 3: *Evaluation of Applied Intra-Disk Redundancy Schemes to Improve Single Disk Reliability* appeared in *Proceedings of the IEEE International Symposium on Modeling, Analysis, and Simulation (MASCOTS)* [GSB+11].
  As discussed in Chapter 1.2, many codes to improve the reliability of a single disk were proposed and analyzed theoretically. In this work, we compare multiple implementations to foster the understanding of the performance of various codes. The work is the first in the field that also adds a detailed view on the energy-consumption overhead of applied codes on both the CPU and disk drive. Also, we contributed and evaluated the *Scrambling* approach that improves simple intra-disk redundancy codes to cope with typical disk drive errors.

- Chapter 4: *LoneStar RAID: Massive Array of Offline Disks for Archival Systems* was submitted to the journal *ACM Transactions on Storage (TOS)* [GNB15].
  This work is an extended and improved version of the RAID scheme presented in [GMPB11]. The work describes the two-dimensional LoneStar RAID scheme. The scheme builds on the improved single-disk capabilities presented in Chapter 3 and is an improvement over the multi-dimensional RAID schemes presented in the introduction in Chapter 1.3. The work presents an analysis of the scheme's reliability and a first evaluation of the system's performance. We present a novel approach to integrate NVRAM into the RAID scheme for metadata journaling and as a parity cache. Multiple parities can be merged which reduces the amount of data written to disks. As a result, energy can be saved by delaying disk writes without sacrificing reliability.

- Chapter 5: *LoneStar Stack: Architecture of a Disk-Based Archival System* appeared in *Proceedings of the IEEE International Conference on Networking, Architecture, and Storage (NAS)* [GBSB14].

In this work, we present the architecture of a file system layer that provides multiple techniques for reliability and recovery and builds on the block-level LoneStar RAID proposed in Chapter 4. With a minimal overhead, erroneous bits can be detected and recovered on both block and file level. The LoneStar Stack provides a file system abstraction that is optimized to save energy. It incorporates semantic data placement strategies, a read cache and a write-through cache. Semantic data placement strategies were described theoretically before [WM10]. We now add a study on its technical feasibility in [GBSB14].

- Chapter 6: The thesis closes with a summary and an outlook of open topics and further research opportunities based on the presented work.

2 | Analysis of the ECMWF Storage Landscape

Abstract

Despite domain-specific digital archives are growing in number and size, there is a lack of studies describing their architectures and runtime characteristics. This paper investigates the storage landscape of the European Centre for Medium-Range Weather Forecasts (ECMWF) whose storage capacity has reached 100 PB and experiences an annual growth rate of about 45%. Out of this storage, we examine a 14.8 PB user archive and a 37.9 PB object database for metereological data over a period of 29 and 50 months, respectively.

We analzye the system's log files to characterize traffic and user behavior, metadata snapshots to identify the current content of the storage systems, and logs of tape libraries to investigate cartridge movements. We have built a caching simulator to examine the efficiency of disk caches for various cache sizes and algorithms, and we investigate the potential of tape prefetching strategies. While the findings of the user archive resemble previous studies on digital archives, our study of the object database is the first one in the field of large-scale active archives.

2.1 Introduction

The number and size of computing sites with domain-specific archives has reached new heights and is still increasing. Next to the ever faster compute systems, storage systems are also growing in multiple dimensions like available capacity, access frequency, and required throughput. With the increased computing power and new algorithms from the big data area, computations tend to use and create more data. Here, many archives become active in the sense that every stored datum may be read at any point in time.

For building storage systems that meet the demands of active archives, it is necessary to understand how today's systems evolved, how they work and in which direction the development is heading. Unfortunately, only a small number of publicly available studies exist that analyze the storage infrastructure and characterize the stored data as well as storage access patterns and growth patterns. The result is a lack of representability of previous studies, as comparable studies are missing. Most of today's multi-petabyte storage systems follow a tape backend + disk caching approach. While disks offer the better performance and more flexibility in their access characteristics, tape is still cheaper in terms of capacity. The disk-to-tape ratio is therefore a tradeoff between price, performance, and capacity.

Previous studies investigated traces of desktop or network file systems [LPGM08, WN13], internet accessible content delivery networks [GALM07, HBvR$^+$13], in-memory caches [AXF$^+$12], or digital archives and content repositories [MAFM12, ASM12, FMR12]. The presented study is the first analysis of an active archive – a large-scale content repository where all data is subject to be accessed at any time.

The contributions of this paper are an in-depth system analysis of two archival systems from the previously uncharted weather forecasting domain, a simulator-driven evaluation of workload trace files to improve the disk cache efficiency, and a feasibility evaluation of tape prefetching strategies. The subject of our study is the storage environment of the *European Centre for Medium-Range Weather Forecasts* (ECMWF)[1], which provides medium-range global weather forecasts for up to 15 days and seasonal forecasts for up to 12 months. To achieve this, they utilize supercomputers[2] and, as of September 2014, storage with a capacity of 100 PB. Next to fast HPC storage, they run two in-house developed archival systems: a general-purpose user-accessible archive (ECFS) for file storage hosting 14.8 PB of data and a large object database for meteorological data (MARS) that hosts 37.9 PB of primary data consisting of 170 billion fields. It is regarded as the world's largest archive of numerical weather prediction data. Both systems consist of multiple tape libraries with disk-based caches in front of them. We have developed a trace-based storage simulator for the ECFS traces to determine the efficiency of various cache strategies and to optimize the hit-ratio of the disk caches. Additionally, we look into the logs of the tape libraries and

[1] http://www.ecmwf.int/

[2] http://www.top500.org/site/47752

the backend HPSS system [ECM14]. Examining logs with more than 9.5 million tape load operations in 2012-2013, we investigate the feasibility of tape prefetching strategies.

Our study shows that the two storage systems are used in different ways. While the ECFS is an archive with mostly write accesses and only a small set of actively used data, the MARS system is read-dominant, and all its data are subject to be read. Both systems face an exponential data increase with a compound annual growth rate (CAGR) of about 45% over the last years and about 50% today. In total, the ECFS logs cover 29 months of 2012-2014 and the MARS logs 50 months of 2010-2014. These logs cover the integration of new applications, models and hardware. Especially, the additional throughput and capacity required by a newly commissioned supercomputer becomes visible at several points and is one of the reasons for the exponential growth in storage capacity.

2.2 Related Work

Large-scale storage and archival systems have been investigated for many years. Baker et al. and Rosenthal et al., for example, discuss the technical and non-technical challenges for building long-term digital repositories [BKM05, BSR+06, RRM+12]. The technical problems include large-scale disasters, component and media faults, and the obsolescence of hardware, software and formats. Furthermore, human errors, loss of data context, or misplanning need to be considered. Rosenthal especially emphasizes the economical aspects of long-term archives. Economic faults, erroneous capacity planning, or the wrong use of storage technologies can be a threat for long-term data availability. Many previous studies help to encounter these threats and to build reliable and successful digital archives.

Most studies that analyze the contents or behavior of file systems deal with workstations, general-purpose network file systems, or HPC storage systems [EK02, ABDL07, LPGM08, DB99, CSGK11, WN13]. They examine static file system snapshots, request traces and operating system logs to investigate multiple dimensions of storage systems. Meister et al. investigated the possible impact of applied deduplication on HPC storage [MKB+12].

Another investigated area are large-scale publicly accessible systems, their usage patterns, and the efficiency of caching [BCF+99, SGD+02, GALM07, AXF+12, HBvR+13]. It is especially important to understand and characterize traffic and user behavior to build and improve caching infrastructures.

There exist only a few publicly available archival traces [Los14] and analyses of recent archives. Madden et al. wrote a technical report on the user behavior in the NCAR archival system over a three year period from 2008 to 2010 [MAFM12]. They also started an investigation of namespace locality of user sessions. Frank et al. compare the logs of an NCAR system to a previous study of the system from 1992 [FMR12]. In the interval, the read-to-write ratio on the system changed from 2:1 to 1:2 which indicates that archives are becoming increasingly write-only. From the traces it was also derived that 30% of the requests have a *latency to first byte* of more than three minutes. In order to improve

the latency, the authors suggest large disk caches that permanently hold the small files. The most comprehensive study was conducted by Adams et al. [ASM12] who examined multiple public and scientific long-term data repositories for their content and workload behavior. Especially for the scientific LANL and NCAR repositories, disks play an increasingly important role, which becomes visible by comparing the 1:262 disk-to-tape ratio at NCAR in 1993 with the 1:3.3 ratio at LANL in 2010.

Previous studies document the rise of disk drives, used either as a complement to or as a replacement for tape in large-scale archival scenarios. Colarelli and Grunwald, for example, argue for the replacement of tape archives by large disk arrays that are switched off when not in use [CG02]. This idea was refined in the Pergamum system by Storer et al., a distributed system of powered-down disks for archival workloads [SGMV08]. Grawinkel et al. proposed a high-density MAID system optimized for "write once, read sometimes" workloads [GBSB14]. Today, large-scale archival systems are in production that primarily build on disk technology, like the Internet Archive [JK09].

2.3 Background

The European Centre for Medium-Range Weather Forecasts (ECMWF) is an independent intergovernmental organization supported by 20 European member states and 14 co-operating states. The center was established in 1975 and hosts one of the largest supercomputer complexes in Europe. Storage for the computation environments is driven by HPC storage systems and two large in-house developed archival systems that will be investigated in this paper. Today the center hosts a combined storage capacity of about 100 PB. This includes the HPC storage and backups. All files of the archival system are stored on tape, and important files are stored to a second tape copy.

ECFS is used as a general-purpose archival system and accessible for users of the ECMWF compute environment. In 2014/09 it stored 137 million files with a total size of 14.8 PB. The system provides 0.34 PB of disk caches so that the disk-to-tape ratio is 1:43. All data is written to a disk cache first before it is migrated to tape.

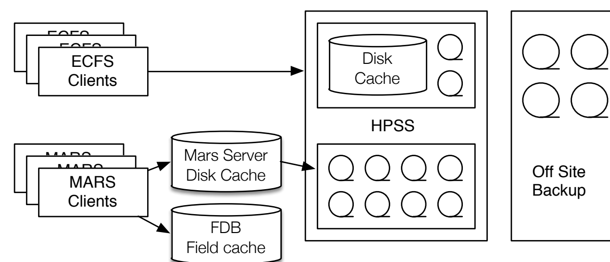


Figure 2.1: Abstract overview of storage environment.

MARS is an object store for meteorological data with a database-like API. A custom query language is used to specify a list of relevant fields. The system assembles these fields into a package and stores it on a target storage system for access from the HPC systems. To reduce the metadata overhead and keep the file backend manageable on tape, fields are stored in appendable files. A dedicated database allocates and maps fields to offset-length pairs on files. New, recently, and often used fields are staged and cached in a dedicated field database (FDB), which is the primary target for queries. If the FDB does not contain a field, the MARS servers are queried. MARS manages its own disk cache and a miss eventually loads (parts of) the required files from tape. All MARS servers for all subprojects provide a total of 1 PB of disk cache which results in a 1:38 disk to tape ratio. The FDBs are stored on the HPC storage systems connected to the supercomputers and can grow to multiple PB. In 2014/09 MARS stored a total of 54 PB of primary data consisting of 170 billion fields in 11 million files. Additionally, the system uses 800 GB of metadata. Each day, 200 million new fields are added.

Figure 2.1 provides a high level overview of the storage environment, which is implemented around the High Performance Storage System[3] (HPSS) that provides the disk caches for ECFS and manages the tape resources for both ECFS and MARS. Tape is considered to be the final destination for data. Every cached file has a copy on tape. In contrast to ECFS, MARS manages its own disk caches outside of HPSS. The ECMWF runs multiple project-specific MARS databases that are mapped to individual storage pools in the HPSS system. In the following, we will treat ECFS and MARS as two different storage systems.

2.3.1 Available Log Files

This work analyzes log files and database snapshots provided by ECMWF. All log files were obfuscated by replacing user information and each part of a file's path by a hash sum, except file extensions. No user information can be revealed, but access patterns and file localities are preserved. We developed a set of python scripts to obfuscate, sanitize, and pack the raw source data. The following analysis is based on compressed log files from multiple ECFS, MARS, and HPSS servers. The gathered and investigated files are:

ECFS access trace: Timestamps, user id, path, size of GET, PUT, DELETE, RENAME requests. 2012/01/02 - 2014/05/21.

ECFS / HPSS database snapshot: Metadata snapshot of ECFS on tape. Owner, size, creation/read/modification date, paths of files. Snapshot of 2014/09/05.

MARS feedback logs: MARS client requests (ARCHIVE, RETRIEVE, DELETE). Timestamps, user, query parameters, execution time, archived or retrieved bytes and fields. 2010/01/01-2014/02/27.

MARS / HPSS database snapshot: Metadata snapshot of MARS files on tape. Owner, size, creation/read/modification date, paths of files. Snapshot of 2014/09/06.

HPSS WHPSS logs / robot mount logs: Timestamps, tape ids, information on full us-

[3] http://www.hpss-collaboration.org/

age lifecycle from access request till cartridges are put back to the library. 2012/01/01 - 2013/12/31.

The traces and tools used are publicly available as outlined in Section 2.9.

2.4 ECFS User Archive

Users of the ECMWF compute environment use ECFS as an intermediate and long-term storage for general purpose data. New and recently retrieved files are stored in disk pools and are migrated to the tape storage by HPSS. Files are categorized by their size and are spread to six pools with different capacities and properties. The ranges as well as the number and size of stored files are listed in Table 2.1. Though tape is considered as the primary storage, files of the *Tiny* class are primarily stored on mirrored disks and only backed up to tape. Therefore, to read tiny files, tape is never used. In ECFS, no files are updated in place, but a file may be overwritten.

Group	From	To (incl.)	Count	Used Capacity
Tiny	0	512 KB	36.0 mil.	4.4 TB
Small	512 KB	1 MB	9.1 mil.	6.3 TB
Medium	1 MB	8 MB	29.5 mil.	101 TB
Large	8 MB	48 MB	30.0 mil.	585 TB
Huge	48 MB	1 GB	29.7 mil.	6.2 PB
Enormous	1 GB	∞	3.1 mil.	8 PB

Table 2.1: File size categorization. Count and capacity refer to Sept. 2014.

2.4.1 Metadata Snapshot Analysis

In contrast to the trace files that only yield data being accessed within the investigated time frame, the HPSS database snapshot gives a full view on all stored files on 2014/09/05.

The summary in Table 2.2 presents a total of 137.5 million files that are stored in 5.5 million directories and use 14.8 PB of capacity. The table also presents the most common file types by count and occupied capacity. Next to the unknown files that do not yield an extension on their file names, packed, compressed, and weather domain specific files are highly represented. Figure 2.2 shows a histogram of the most common file sizes. The system stores a large amount of files between 0 bytes and 1 KB, but else visually follows a Gaussian distribution that peaks at 8-16 MB.

The database excerpt also contains the *creation*, *modification*, and *read* timestamps of files. These timestamps mark the access times of a file on the tape drives and do therefore not reflect the access times of cached files. Files of the *Tiny* group (see Table 2.1), for example, are fully cached on disk and never retrieved from tape. The file system statistics of Table 2.2 show 101.3 million files that were never read from tape. These files were only uploaded or modified. If they were accessed, they were read from the HPSS' disk cache.

File system stats	
Total #files	137.5 mil.
Total used capacity	14.8 PB
Largest file size	32 GB
#Directories	5.5 mil.
Max files per directory	0.43 mil.
#Files never read from tape	101.3 mil. (11.3 PB)

Most common file types	
by file count	**by used capacity**
unknown (27.8232%)	unknown (39.3306%)
.gz (20.4319%)	.tar (21.2699%)
.tar (7.8015%)	.gz (12.4954%)
.nc (7.6312%)	.nc (7.8819%)
.grib2 (1.9438%)	.lfi (2.2399%)
.raw (1.7284%)	.pp (1.0087%)
.txt (1.5095%)	.sfx (0.9327%)
.Z (1.4862%)	.grb (0.8471%)
.bufr (1.4451%)	.grib (0.3977%)
.grb (1.4402%)	.bz2 (0.3083%)

Table 2.2: Statistics on ECFS tape storage.

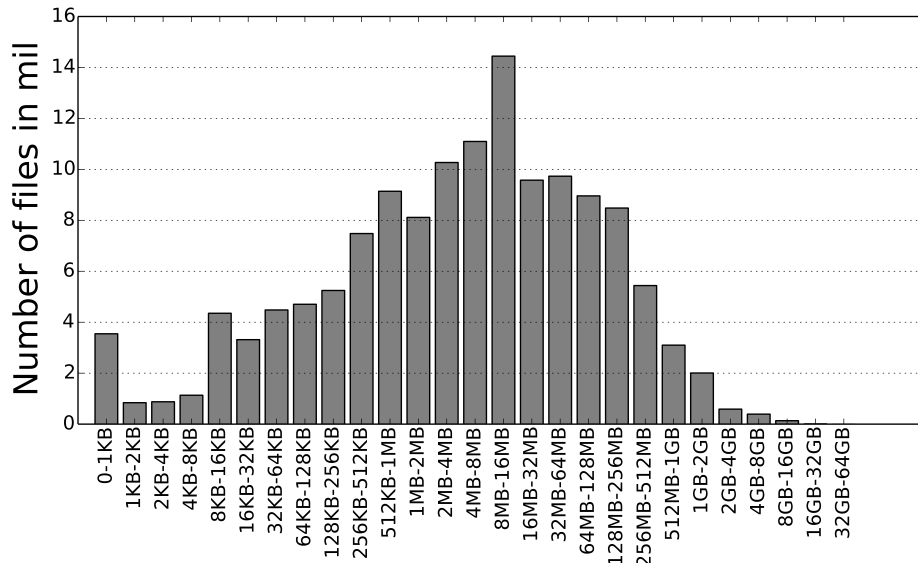


Figure 2.2: Histogram of stored ECFS files sizes.

The upper graph of Figure 2.3 visualizes the absolute number of existing files at a particular point in time and the number of files that were *unread* or *unmodified* since that date. An *unaccessed file* was neither read nor modified and an *existing never read file* has no read time stamp and was therefore created and possibly updated only. The lower graph

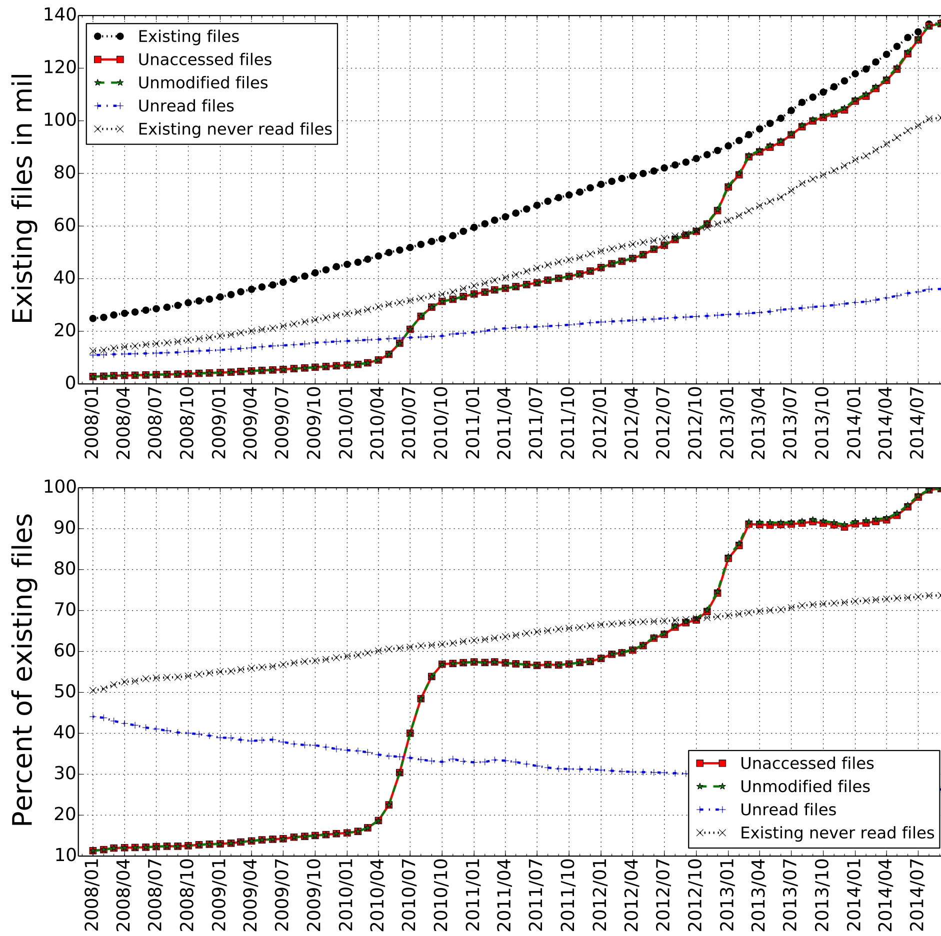


Figure 2.3: Top: Total number of existing files with amounts of unmodified and unread files since the point in time. Bottom: Fraction of unread or unmodified files relative to existing files.

presents the fractions relative to the existing number of files to delineate the behavior over time. Since 2012/01 60 million new files were created. With the beginning of 2013, the new supercomputer becomes visible as more files are created and the graph steepens. In general, the number of existing files follows an exponential growth. The number of unmodified files closely follows the number of unaccessed files. This means that most of the last actions on files were modifications, which also includes the initial creation, and not reads. In the last year (2013/07 - 2014/07), the amount of never read files slightly grew from 68% to 73%, while the number of unread files shrank from 27% to 25%. In 2013/07 about 90% of the data stored on tape were neither read nor modified. This value changes by looking back an additional year till 2012/07 which covers the introduction of the new supercomputer. About 64% of the files existing at that point in time were not accessed until 2014/07.

2.4.2 Workload Characterization

For the time period from 2012/01/01 to 2014/05/20 a full trace of all ECFS operations has been investigated. Table 2.3 summarizes the key metrics.

Total GET requests	38.5 mil.
Total GET bytes	7.24 PB
Total PUT requests	78.3 mil.
Total PUT bytes	11.83 PB
Total DEL requests	4.2 mil.
Total RENAME requests	6.4 mil.
Total different files	73.4 mil.
Total different dirs	6.2 mil.
#Files with PUT	66.2 mil.
#Files with GET	12.2 mil.
Cache hit ratio by requests	86.7%
Cache hit ratio by bytes	45.9%

Table 2.3: Characterization of ECFS workload 2012/01/02 - 2014/05/21.

During the observed timespan, a total of 38.5 million GET requests were executed on 12.3 million unique files, a total of 78.3 million PUT requests on 66.2 million unique files were counted, and 4.2 million files were deleted from ECFS. As there are more unique written files than PUT requests, some files were updated or overwritten. In comparison to the NCAR analysis [FMR12] where 30% of the stored files were read during the 29 month observation timespan, ECFS saw reads on 12.3 million distinct files which is 9% of the total corpus.

We analyzed the number of PUT and GET requests and their respective traffic based on the ECFS file size categories of Table 2.1. Figure 2.4 breaks down these metrics on a monthly basis. Until 2013/03, the system has a balanced throughput of 200-300 TB PUT and GET traffic per month with slightly prevailing write traffic. With the introduction of

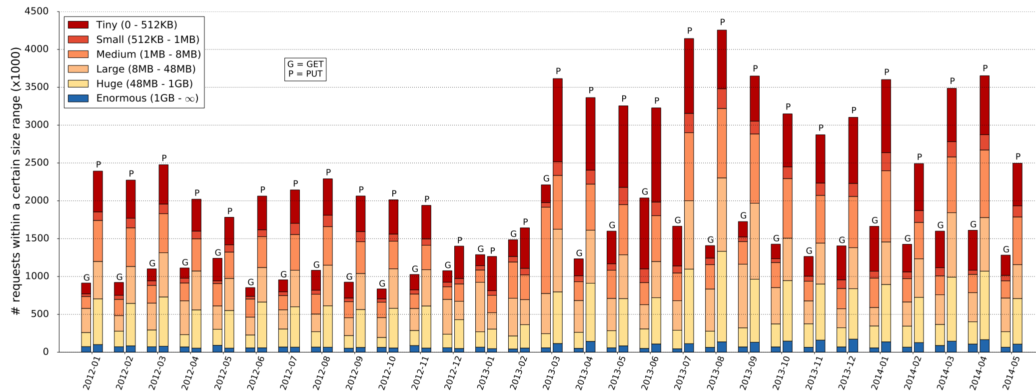


Figure 2.4: Total requests per month

the new computer in the first quarter of 2013, the amount of written data doubles both, traffic- and request-wise, while the retrieval rates and volume merely rise.

Figure 2.5 visualizes the hotness of files during the observed timespan. As the analysis in Section 2.4.1 shows, the system contains a lot of data that have not been accessed within this time frame or were just uploaded without being retrieved again.

The main argument that is underlined by the plot is that in total 50% of all GET requests hit less than 5% of the stored files. This argument is supported by the overall 86.7% disk cache hit ratio with a disk-to-tape ratio of 1:43. The cached requests are responsible for 45.9% of the retrieved bytes, which leads to the assumption that most small reads can be satisfied by the cache and mostly larger files are retrieved from tape. The later cache analysis in Figures 2.15 and 2.16 (see Section 2.7) visualizes the cache hit ratios observed from the ECMWF traces for the different file size categories. While *Tiny* files achieve 100%, only 60% of the *Huge* and 50% of the *Enormous* file retrieval requests are served from disk.

2.4.3 User Session Analysis

For every request in the trace, the user id and the host that executed the command are known. A user id can be used by all sorts of processes running on multiple systems at the same time. The trace reveals 1,190 unique user ids and 2,647 unique hostnames. As presented in Table 2.3, a total of 11.8 PB of data have been written and 7.2 PB have been retrieved from the archive. Figure 2.6 visualizes how the bytes and requests are distributed to the identified users. The plot shows that only 850 users wrote data while 1,075 users retrieved data. Furthermore, only a small fraction of less than 100 users make up more than 90% of the traffic.

We gathered all commands issued by a user id from the same hostname and create clusters of executed requests that occurred within close succession. If the time between two requests is longer than the window, they are clustered into different groups. We call all requests within such a group a user session. Figure 2.7 presents the number of actions

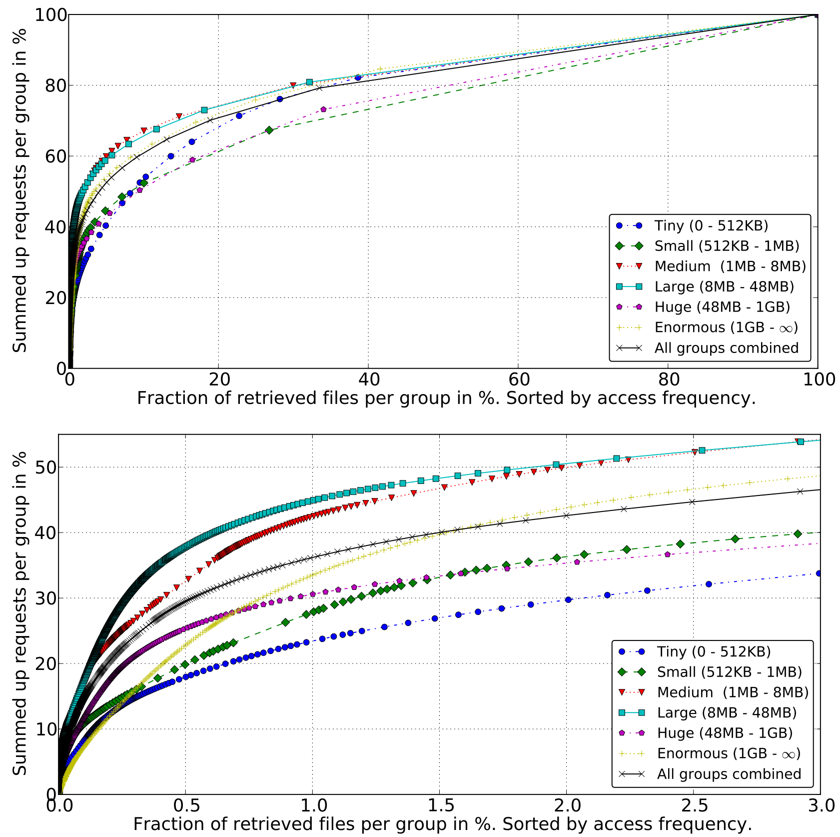


Figure 2.5: Top: CDF over file GET requests per file per size group.
Bottom: Zoomed to most frequently retrieved 3 %.

Key	P05	P50	mean (+-95%)	P95	P99	Count
#Sessions per user_id	1	35	2,276.76 ($\pm$ 1,006.85)	7,315	28,861	1,190
#Sessions per user_id@host	1	4	92.70 ($\pm$ 7.50)	352	1,512	29,227
Total #Actions	2	7	47.04 ($\pm$ 0.69)	126	579	2,709,343
GET Requests per session	1	4	35.55 ($\pm$ 0.78)	108	571	1,083,067
ReGET requests per session	1	2	31.98 ($\pm$ 3.66)	99	442	132,515
PUT Requests per session	1	5	34.43 ($\pm$ 0.44)	97	373	2,274,645
Dirs with GETs	1	2	8.15 ($\pm$ 0.16)	21	96	1,083,067
Dirs with PUTs	1	2	6.06 ($\pm$ 0.07)	14	71	2,274,645
Retrieved files per directory	1	1.78	10.05 ($\pm$ 0.23)	30	149.75	1,083,067
Archived files per directory	1	2	7.44 ($\pm$ 0.12)	27	74	2,274,645
Retrieved MBytes	0.56	192.13	7,172.91 ($\pm$ 221.74)	17,150.69	86,444.15	1,083,067
Archived MBytes	0.02	206.04	5,591.52 ($\pm$ 94.84)	19,588.74	64,995.07	2,272,399
Session lifetime in s	0	154	2,601.60 ($\pm$ 21.13)	9,295	38,456	2,709,343
Gap between sessions	120	250	896.26 ($\pm$ 11.70)	3,070	3,500	29,227

Table 2.4: ECFS user session analysis. A total of 2,709,343 sessions were identified.

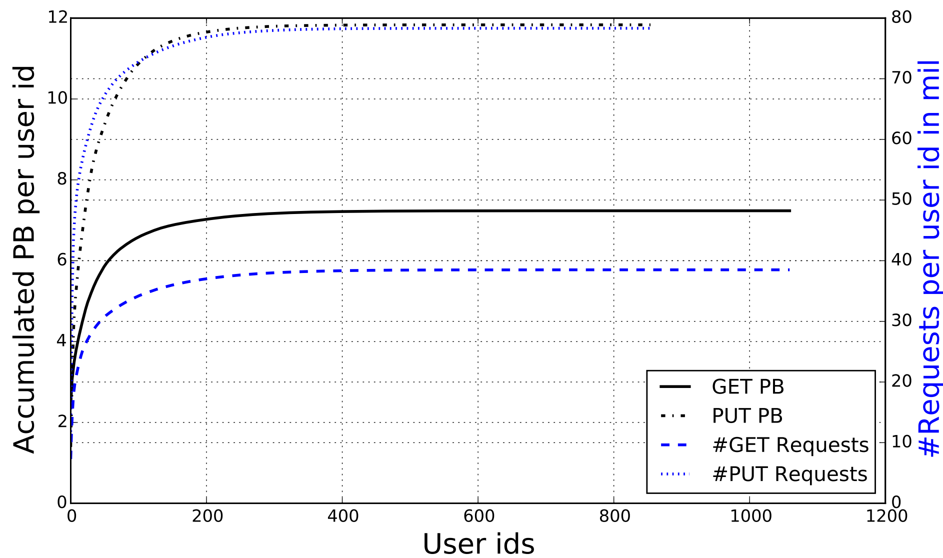


Figure 2.6: Summed up traffic for GET and PUT requests per unique user id

per identified user session for a growing time window based on the methodology used by Madden et al. [MAFM12]. In contrast to Madden's approach, the graph does not show a plateau that would characterize a typical session. Therefore, we used a machine learning approach over all actions of each user to identify a window size that produces the most stable clusters for that user. For the 1,190 users, we identified a total of 2,709,343 sessions and Table 2.4 presents statistics over some key performance points. As for all following statistics, we present the 5, 50, 95 and 99 percentiles, a mean with a 95% confidence interval and the count of occurrences of the performance points. The statistics for a metric like *GET requests per session* is only counted if the session had at least one GET request. A write only session would not be counted here.

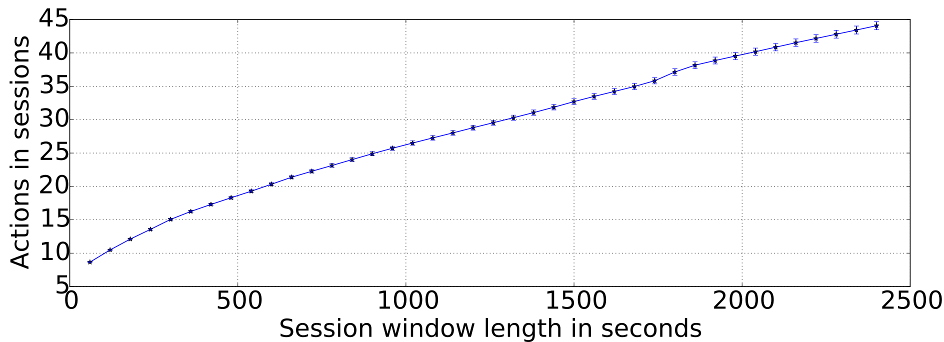


Figure 2.7: Mean actions per user sessions for growing window sizes.

The results underline the trend shown in Figure 2.6. A small fraction of users is very active which becomes visible in the high differences of the P50 and P95 percentiles of multiple performance points. Also the volume in terms of requests and transferred bytes follows this trend, where a small fraction of sessions dominate the workload. The session lifetime considers the time from the beginning of the first until the end of the last action within a user session. Many sessions consist of a single action that is executed within a second or less. On the other hand, we observe sessions longer than 10 hours for the P99 which are most probably regular operations like cron jobs for backups.

The trace file yields full obfuscated paths of accessed files. Therefore, we investigated the locality of accesses within a session. If a session retrieved files, on average 36 GET requests were issued which accessed 8 directories with about 10 files per directory. If a session also archived data, on average 34 files were uploaded to 6 different directories with about 7 files per directory. As observed in the study by Adams et al. [ASM12], we also see user sessions that re-retrieve the same file within the lifetime of the session, which occurred in 132,515 of the 1,083,067 sessions with GET requests. Out of the total 38.5 million GET requests, 11% (4.2 million) were re-retrievals of files within a user session.

2.5 MARS Database

The Meteorological Archival and Retrieval System (MARS) is the main repository of meteorological data at ECMWF. It contains petabytes of operational and research data, as well as data from special projects. In contrast to the ECFS, where files are identified by a unique path, MARS hosts billions of metereological fields that cannot be directly addressed by a user, but are the result of a query. The available log files of the MARS system contain all parameters of the queries and the number and source of the returned fields, but do not allow to identify the exact keys of the accessed fields. Therefore, this analysis cannot investigate the hotness of fields or files, but can only characterize the observed traffic.

MARS is based on a 3-tier storage architecture with the FDB as the first, the MARS servers' disk caches as the second and the HPSS tapes as the third layer. All fields in MARS are eventually persisted to the files in the HPSS tape backend, but requests are primarily served by the FDB and the MARS servers. The system also applies domain specific knowledge to improve the cache hit rates. For example, if files or full tapes are identified as hot, they can be manually loaded and locked to the MARS servers' disk caches. Currently 250 TB are reserved for this manual cache optimization.

2.5.1 Metadata Snapshot Analysis

The following analysis investigates an HPSS database snapshot of all MARS files on tape from the 2014/09/04. Table 2.5 presents a summary of the findings. Compared to ECFS, MARS stores a significantly smaller amount of files that use a larger total capacity of 37.9 PB. When the snapshot was taken, a total of 7.8 million of the 9.7 million stored files were never read from tape.

File system stats	
Total #files	9.7 mil.
Total used capacity	37.9 PB
Largest file size	1.34 TB
#Directories	555,799
Max files per directory	38,375
#Files never read from tape	7.9 mil. (24.9 PB)

Table 2.5: Statistics on MARS' tape storage.

Similar to ECFS, the histogram of the file sizes in Figure 2.8 resembles a Gaussian distribution, yet with a higher average file size and a higher maximum at 128-256 MB. The size of the largest file stored is 1.34 TB.

The visualization of *creation*, *modification*, and *read* times in Figure 2.9 follows the schematic described in Section 2.4.1. Again, the upper graph visualizes the absolute number of existing files at a particular point in time and the number files that have not been modified or read since that date. The lower graph presents the fractions relative to the existing number of files. The high rate of unaccessed files and a modification rate close to 100% shows that files are predominantly created, rarely updated and only read sometimes from the HPSS tape backend. The lower graph shows that up to 80% of the files on tape were written, but never read again. This behavior either indicates a cold storage or a strong caching infrastructure. In contrast to the ECFS analysis (see Figure 2.3), the introduction of the new supercomputer in Q1/2013 is not visible in Figure 2.9. Though Figure 2.10 shows a significant change of daily written fields and bytes, the file creation rate on the HPSS system does not change. This is because new fields are aggregated at the FDB and MARS server levels that are written as a file which then appears as a new file in HPSS. The assumption is that the average size of newly created files grows over time.

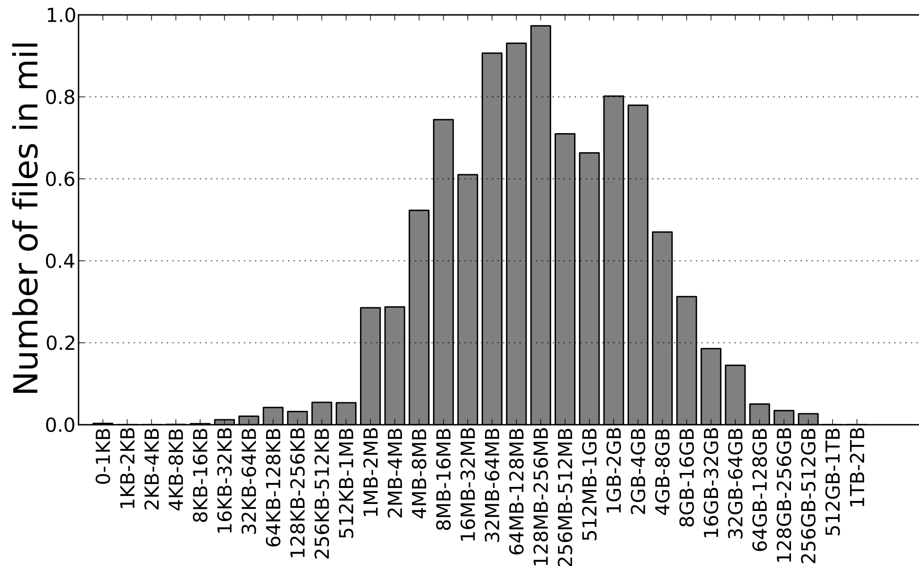


Figure 2.8: Histogram of files sizes

2.5.2 Workload Characterization

We present the analysis of the MARS feedback logs over the timespan 2010/01/01-2014/02/27 that quantifies user requests and the resulting traffic. Figure 2.10 visualizes the daily throughput in bytes and requests. Again, the introduction of the new compute environment in Q1/2013 becomes visible in elevated throughput rates. The old environment can be characterized by a constant daily read rate of 40-50 TB (100-120 million fields) and 15-20 TB (about 50 million fields) of written data. With the new computer, the read rate doubles, but the write rate nearly increases threefoldly. During the peak throughput rate in 2013/01 both old and new supercomputers were running.

Table 2.6 presents some key characteristics of the considered 50 months. A total of 1.2 billion read requests were executed that fetched 269 billion fields which accounted for 91.6 PB of data, while 115 million requests created 114.7 billion new fields which account for 35.9 PB data. The logs breakdown each request into the number of fields, their summed up sizes and source of the returned fields. In total, 80.7% of all requests can be fully served by the field database (FDB), 17.6% of the requests also require data from the MARS server's disk drives and only 2.2% of the requests include data from tapes. 1.4% of the retrieve requests are fully served from tape without any cached data from disk. Considering the number of retrieved fields, MARS achieves a 95.1% cache hit ratio with a 1:38 disk-to-tape ratio on the MARS servers and a similar but not concretizable ratio on the FDB.

Figure 2.11 characterizes the most active user ids in terms of retrieved and archived bytes and the according requests. It shows that the number of users who created content is significantly smaller than the number of users who retrieved data. A huge amount of

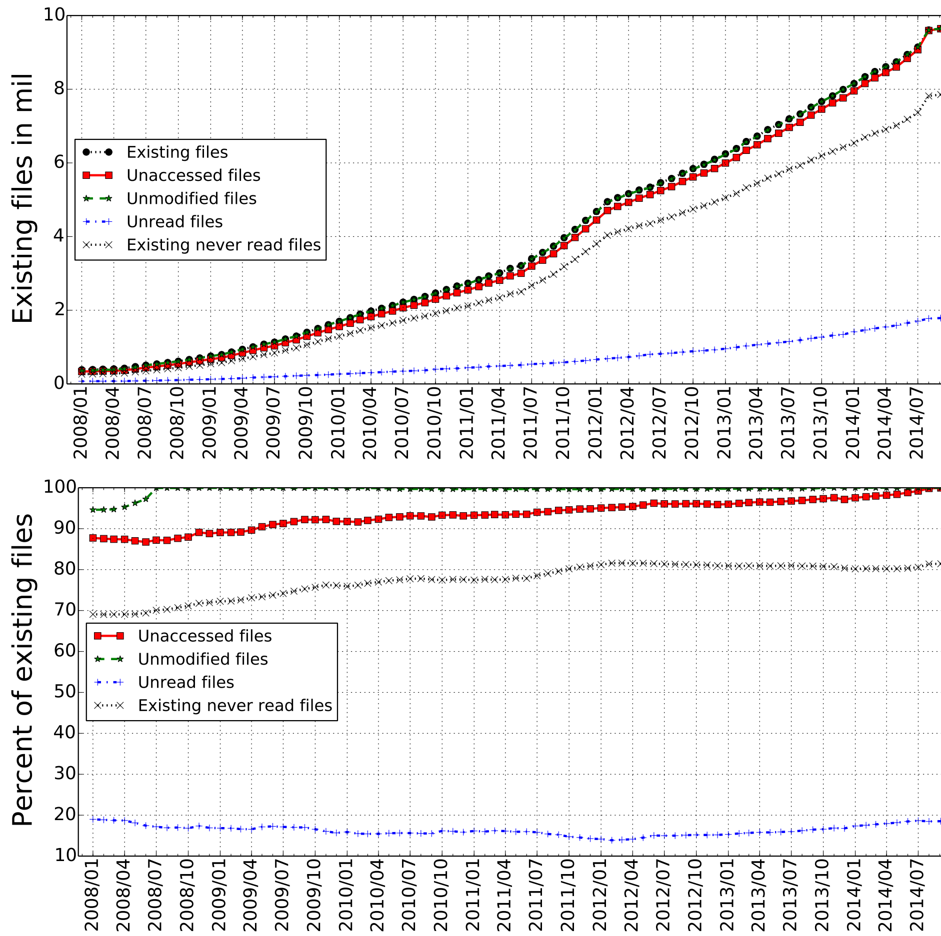


Figure 2.9: Top: Total number of existing files with amounts of unmodified and unread files since the point in time. Bottom: Fraction of unread or unmodified files relative to existing files.

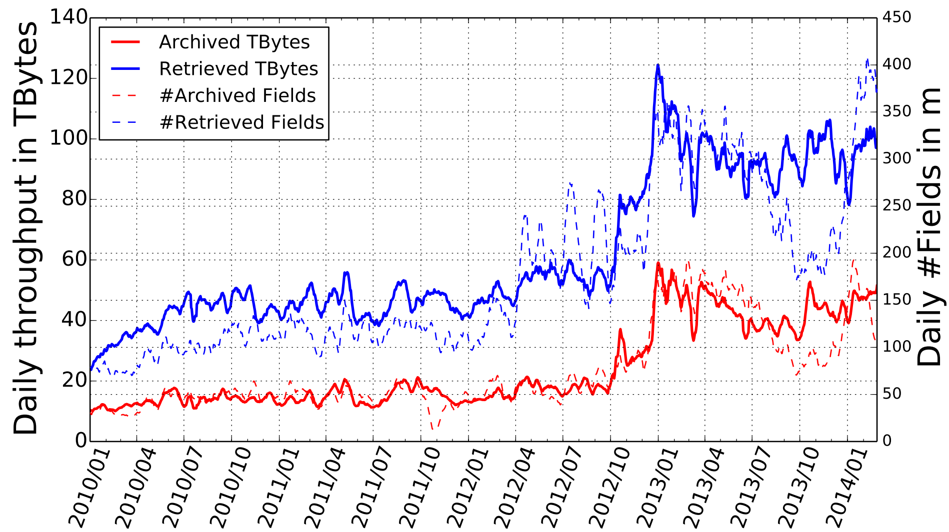


Figure 2.10: Throughput and number of accessed fields.

traffic was actually generated by only two user ids. Some ids are shared by multi-purpose users, behind which multiple real user ids hide. Therefore, these very active users have to be considered as outliers.

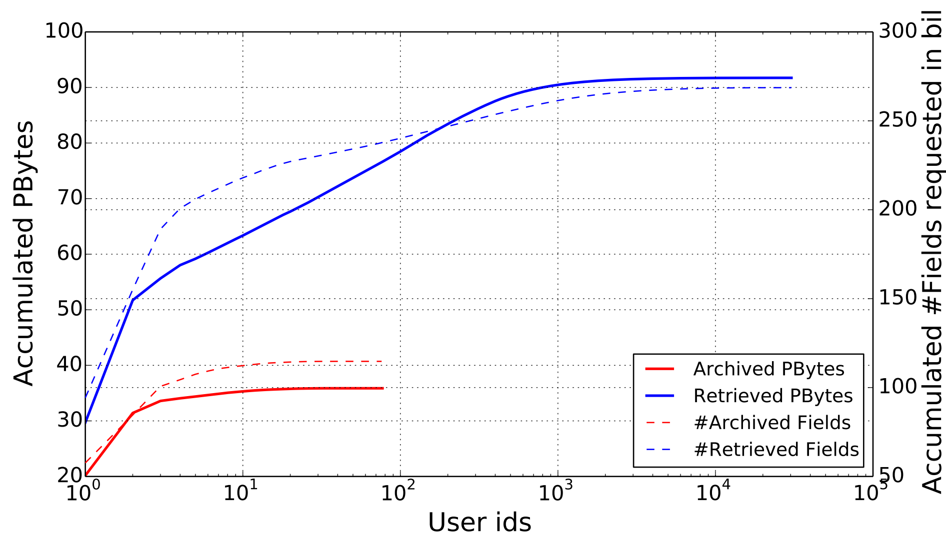


Figure 2.11: Traffic per unique user id.

Total retrieved bytes (fields)	91.6 PB (269 bil.)
- from FDB bytes (fields)	54.2 PB (212 bil.)
- from MARS/disk bytes (fields)	29.4 PB (43.3 bil.)
- from HPSS/tape bytes (fields)	8 PB (13.3 bil.)
Total retrieve requests	1.2 bil.
- including FDB	992 mil. (85.3 %)
- from FDB only	938.9 mil. (80.7%)
- including MARS/disk	204.9 mil. (17.6%)
- from MARS/disk only	151.3 mil. (13%)
- including HPSS/tape	25.3 mil. (2.2%)
- from HPSS/tape only	16 mil. (1.4%)
Total archive requests	115 mil.
Total archived bytes (fields)	35.9 PB (114.7 bil.)

Table 2.6: Characterization of MARS workload 2010/01/01-2014/02/27.

2.6 Tape Mount Logs

Both ECFS and MARS use a HPSS powered tape archive as the final destination and primary copy of files. Despite the strong caching infrastructure, the tape robots are heavily used. Figure 2.12 illustrates the life cycle of one tape use in a simplified state diagram. To

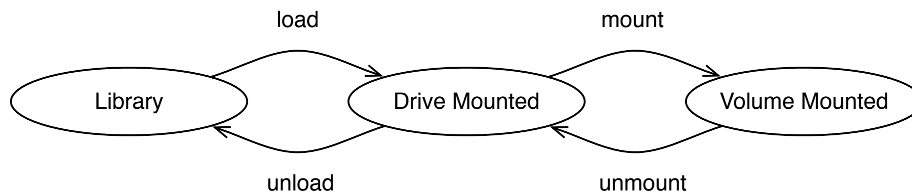


Figure 2.12: Tape states

read data, a request is sent to the HPSS system that *loads* the tape into a drive. The tape is *mounted* as a volume and can be accessed. After a fixed timespan or on request, the volume is *unmounted*, but the tape can remain in the drive. If the tape is requested again while still being loaded, it can be *remounted*, or in the worst case *reloaded* into another drive. Eventually the tape is *unloaded* from the drive and put back to the library. Each tape has a unique identifier that indicates its type (STK T10k-B/C/D) and is assigned to ECFS or a MARS project. We use the identifiers to track the usage of the cartridges and map them to either ECFS or MARS. During the complete log period we saw a total of 231 different drive identifiers and 9,594 unique tape ids for ECFS and 23,118 for MARS.

Figure 2.13 presents the access frequencies of tapes. Sorted by the most often accessed tapes, the total number of loads is summed up. The graph shows that MARS is accountable for 6.7 million and ECFS for 2.8 million tape loads. The right graph of the figure compares

Tape mount frequencies							
System	**#Tapes**	**P05**	**P25**	**P50**	**mean (+-95%)**	**P95**	**P99**
MARS	23,118	0	2	46	291.22 (± 9.70)	1,106	3,351
ECFS	9,594	1	12	85	296.64 (± 11.18)	1,408	2,470
Tape mount latencies in seconds							
System	**#Mounts**	**P05**	**P25**	**P50**	**mean (+-95%)**	**P95**	**P99**
MARS	6,730,218	26	30	35	54.35 (± 0.06)	155	262
ECFS	2,845,154	25	28	32	48.19 (± 0.07)	138	257

Table 2.7: Tape mount statistics

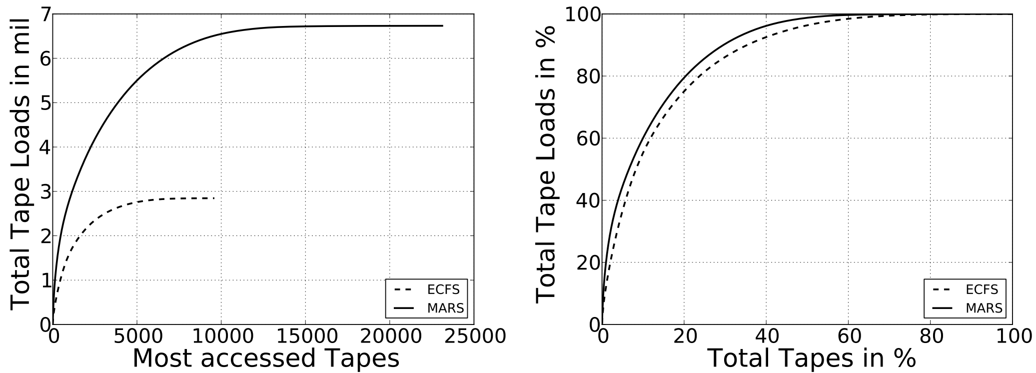


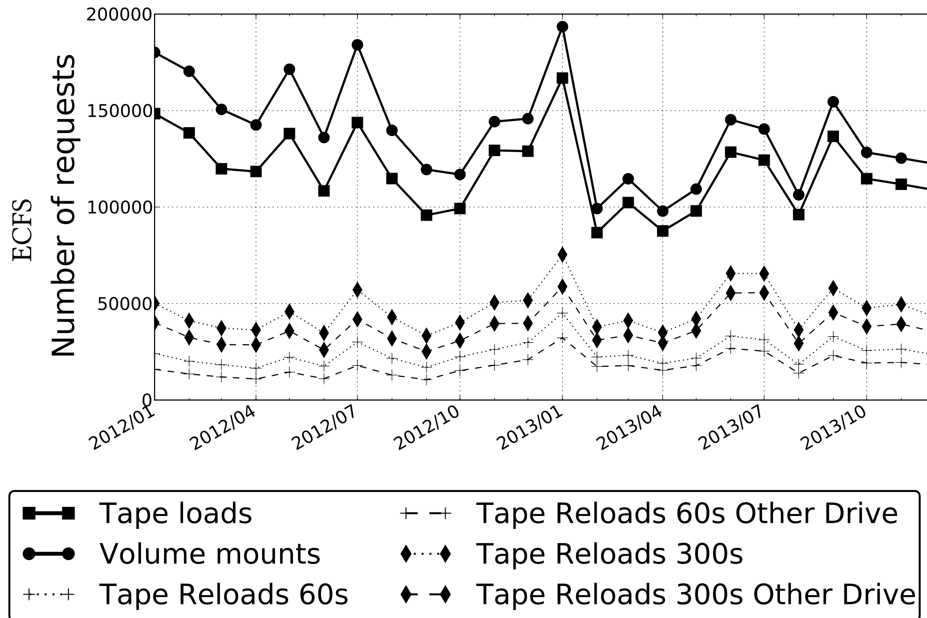
Figure 2.13: CDF over load requests per tape cartridge. Left: Absolute. Right: Normalized.

MARS and ECFS and reveals a similar distribution pattern. About 20% of all tapes are accountable for 80% of all mounts and more than 50% of the tapes are accessed in less than 5% of the loads.

Next we investigate the behavior of the tape system over time. Both, the robot mount logs and the WHPSS logs were used for the analysis. Unfortunately, 6 days of the WHPSS logs were erroneous. We used the logs to feed finite state machines to track the cartridge states. Table 2.7 presents statistics of the time till a requested tape is available for work (volume mounted) and the number of load requests.

The ratio of 6.7 million MARS loads to the 2.8 million ECFS loads perfectly reflects the ratio of the stored data of 35.9 PB to 14.8 PB. The tape load times for the two categories are very similar, which leads to the assumption of equal or shared hardware in the backend. Although the median waiting time is only 35 (32) seconds, the mean is much higher due to some very long waiting times. More than 5% of all tape loads take more than 2 minutes and 1% of the loads take longer than 4 minutes.

Figure 2.14 visualizes the HPSS behavior for ECFS and MARS over time. The graphs show more volume mounts than tape loads, which shows the fraction of remounts without tape movements. The bottom of the graphs present the number of tape reloads and volume remounts within 60 and 300 seconds. A *tape reload 60s* means that after the tape was unloaded, within 60 seconds another mount request was issued. Over the observed time



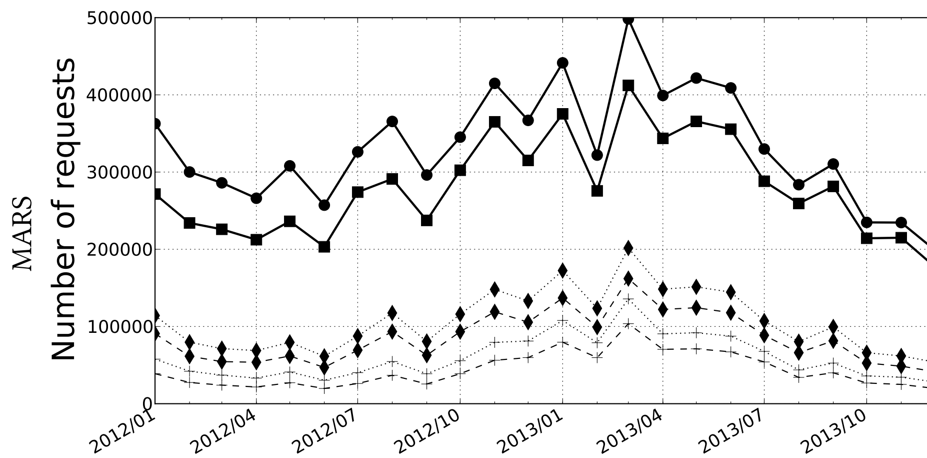


Figure 2.14: Mount Details for ECFS and MARS

Time slot (s)	MARS	ECFS
(0, 60]	73.922	8,740
(60, 120]	48,580	8,270
(120, 300]	107,276	23,126
(300, 600]	129,861	31,145
(0, 600]	359,639	71,281
(600, 1200]	189,691	46,792
(1200, 1800]	146,249	36,733
total success	695,569	154,806
[0, -1800]	849,229	162,965
(1800, fail]	8,061,777	1,364,171
Positive hit rate in %	7.24	9.20
Neutral hit rate in %	8.84	9.69
Total misses in %	83.92	81,11

Table 2.8: Prefetching hits based on the correlation analysis

frame ECFS and MARS show a total of 21% of reloads within 60 seconds and 39% of reloads within 300 seconds. The significant finding is that in total 14.8% of all loaded tapes were unloaded from another drive less than 60 seconds ago.

2.6.1 Tape Prefetching

The average loading time could be improved by prefetching the tapes which are likely to be read next. In order to identify such tapes, we performed a correlation analysis on the tape mount logs using the Pearson correlation. In the following we describe the procedure and estimate the potential for improvement.

For any combination of two tapes x and y with a correlation coefficient of at least 0.8, we analyzed all load requests of x and measured the time difference until y was requested. Assuming the system would prefetch y once it sees a request for x, then this delay indicates the time y occupies a drive until it is requested. Table 2.8 shows the number of load requests within different time slots for the ECFS and MARS tapes.

We consider access delays of more than 1,800 seconds as prefetching failures, because with a high probability the tape would be evicted before being accessed. Furthermore, the interval [0, -1,800] shows the number of operations that did not result in a hit, but saw a load of y within the preceding 30 minutes. This is the case if x and y are requested at the same time, if y is already loaded or y was requested prior to x. In this case x would be the prefetching result of y's load request and therefore, should not issue a prefetching event itself. This time slot neither generates any profit, nor induces any costs, which is why we call these events neutral hit and do not consider them as misses. Misses are load requests of x that never see a corresponding load event of y during the following 1,800 seconds.

The MARS and ECFS logs show a total of 695,569 and 154,806 prefetching hits which on average would have resulted in a hit every 74 seconds. Considering the mean latency

of 54.35 and 48.19 seconds per load request, the latency of these operations would have accumulated to 1.20 and 0.24 years and could be saved by prefetching. The total load request latency of all load operation of the two projects is 11.60 and 4.35 years, respectively. The above mentioned reduction of 1.20 and 0.24 years could reduce these by 10.3% and 5.5%. This is a theoretical upper limit, since the 9.4 million misses would nearly double the amount of tape loads and clearly is unsuitable.

To design a prefetching strategy, possible candidates have to be identified. Furthermore, it has to be verified that the robots and drives have enough spare resources to process the prefetching load operations without impairing operational use. The logs show an average of 546.33 ($\pm$ 3.65) load operations per hour and during the busiest 5% of hours, more than 894 operations were performed. During the peak 1% utilization, more than 1,046 load operations were executed per hour. In the absence of such peak loads, the robots should be able to handle additional loads induced by prefetching misses. Finally, prefetching would not be applicable if the drives are constantly busy. We consider a drive to be idle if a tape is loaded but not mounted. Since mounted tapes are always loaded, the ratio of the volume mount time to the tape loaded time is a good metric for the drives utilization. We calculated this ratio for every hour of the observed time frame and on average see that tapes are accessed 82% of the time they are loaded. For the 0.01 and 0.99 percentiles we see a usage of 65% and 94%, respectively. This shows that the drives are highly utilized, but offer idle time to apply prefetching strategies.

2.7 Cache Simulation

Using a newly designed cache evaluation environment, different cache eviction strategies have been analyzed running the ECFS trace files (which were described in Section 2.4.2). We reuse the file size categorization of Table 2.1 and investigate the cache efficiency for different caching strategies and cache sizes. All GET, PUT, DEL, RENAME operations of the trace are replayed to a simulator that mimics a simple disk cache. A GET request on a non-existent file triggers a cache miss and the file is loaded to the cache. Also all PUT requests load the file into the cache, which might lead to an eviction. We evaluate the following cache eviction strategies using the ECFS traces that cover a timespan from 2012/01 till 2014/05 and are visualized in Figure 2.4.

LRU Data is evicted on a Least-Recently-Used strategy.

MRU Data is evicted on a Most-Recently-Used strategy.

FIFO Queue based eviction.

RANDOM A random cache entry is chosen for eviction. The presented graphs show the average results over 10 runs.

ARC Adaptive Replacement Cache that keeps track of both frequently and recently used files with an eviction history [MM03].

Bélády Adaption of the Bélády algorithm [B́66] which evicts those files that will not be needed for the longest time in the future. This algorithm would only be optimal if all files had the same size, but nevertheless we use this almost perfect cache as a baseline. The construction of an optimal cache is NP hard [CWMX10].

The results are visualized in Figures 2.15 and 2.16. For every file size category used at ECMWF (see Table 2.1) we present a graph that analyzes the cache hit ratio for the different caching strategies and multiple cache sizes. The last row shows the relative difference for a single *combined* cache for all files against the combined hit ratio over the sum of all hits and misses of the 6 subcaches. Its capacity steps are the sum of the same step of the other six caches. A negative result means that a single huge cache has a better hit ratio in terms of requests. We used the full year of 2012 to warm up the caches and present the total cache hit ratio for the period from 2013/01 to 2014/05.

Only the ECMWF baseline for *Tiny* files achieves a 100% hit ratio, as all files are always held on disk. The first GET request on a file that had no previous PUT request in the observed trace will result in a cache miss. Therefore, our results cannot reach 100%. The rightmost capacity of the graphs present a cache with unlimited size which never evict files because it can hold all of them. Therefore, this point presents the theoretical maximum cache hit ratio for the observed time frame.

The graphs show that strategies like *MRU* and *FIFO* are not usable at all. Only for very large caches, they yield an acceptable hit ratio. Due to the high number of re-GET requests, the most recently used files should be cached and not evicted, which explains the bad hit rates for *MRU*. Also the *FIFO* reveals a bad performance because it neglects the popularity of files. The constant writes of new files will evict files independent of their usage patterns.

While *MRU* and *FIFO* show a harmful behavior for cache efficiency, the *Random* strategy provides an unoptimized baseline. The *LRU* strategy accommodates the observations of the user sessions and the hot files presented in Figure 2.5 as it does not evict recently used files. The *ARC* cache competes as an improved LRU and in general slightly outperforms the *LRU* strategy.

The *Bélády* provides the best results, but as a theoretical construct that requires knowledge of the future we can only use it as an upper limit. Even for small cache sizes, this strategy often reaches the maximum hit ratio. This observation creates the assumption that an anticipatory eviction strategy that learns from the past might outperform the other strategies. Domain knowledge as observable from the user sessions presented in Section 2.4.3 is available and could be fed to the caches.

Interestingly, the analysis of a big combined cache reveals that up to a cache size of about 2 PB, the single big instance in all cases provides a better hit rate. For a total cache size larger than 2 PB, the 6 subcaches provide better results for all cache strategies. All visualized hit ratios follow an upward trend for more capacity. The simulator can help to identify the achievable improvement of the hit ratio for extra cache capacity.

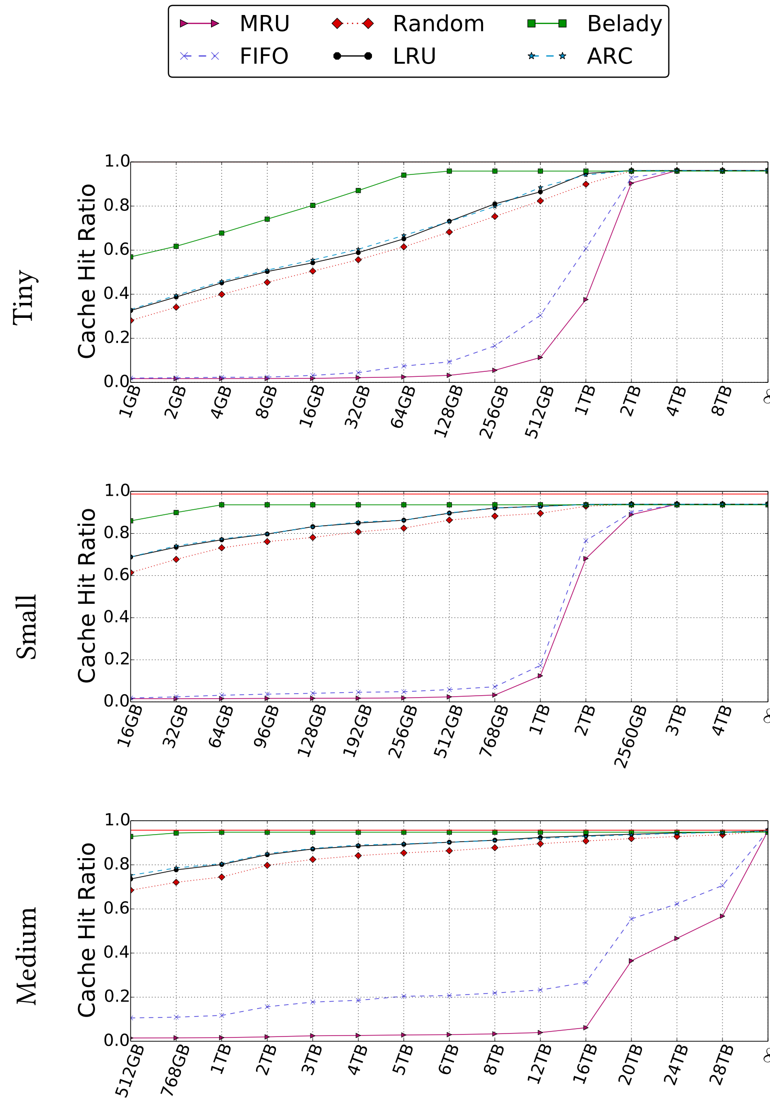


Figure 2.15: ECFS Cache hit ratio evaluation. 2012 is used for cache warm up. Ratios are measured for 2013/2014. Horizontal red line marks ECMWF's hit ratio.

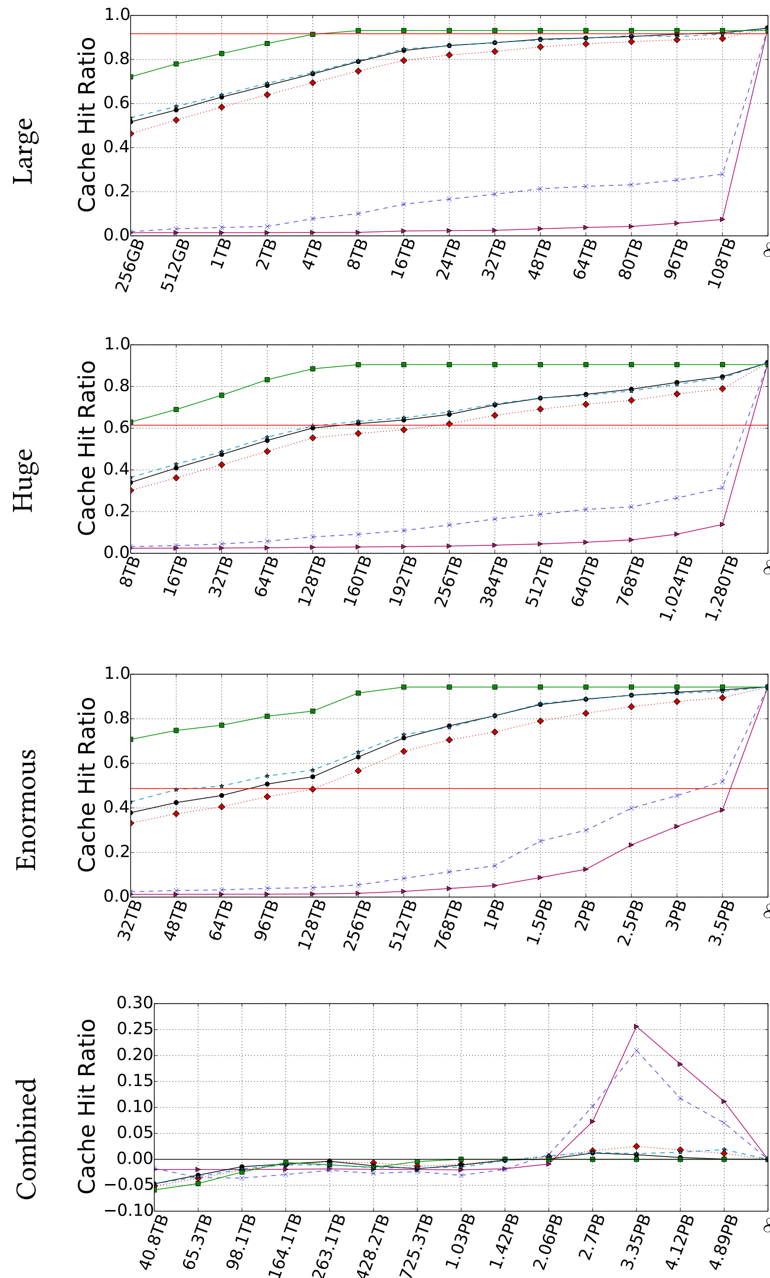


Figure 2.16: ECFS Cache hit ratio evaluation. 2012 is used for cache warm up. Ratios are measured for 2013/2014. Horizontal red line marks ECMWF's hit ratio.

2.8 Discussion & Conclusion

This work analyzed log files and database snapshots to understand the behavior of ECFS and MARS, the two main archival systems at ECMWF, including the tape libraries that form the storage backbone of the two systems. The ECFS system resembles a typical archive and our findings underline the characterizations of previously studied systems [ASM12, MAFM12]. We analyzed the caching infrastructure of ECFS and provide a model and simulator to compare caches with different strategies and capacities. While ECMWF uses a lot of domain-specific knowledge which cannot be described algorithmically, we used the simulator to test basic strategies. It turned out that the *Adaptive Replacement Cache* (ARC) which evicts the *least recently used* and *least frequently used* files from the cache performs best. This conforms to our observation that files are often retrieved again after a short while (usually in the same user session). Coupling our test results with the knowledge of ECMWF will help in the future to further improve their cache hit rates.

Unlike ECFS, MARS opens the uncharted category of active archives that has not been thoroughly investigated until now. The object database uses a three-tiered architecture and a custom query language. All stored fields are subject to be read by computational models or experiments at any time. The investigated log files do not provide all the information necessary to fully characterize the user traffic. It is, for example, not possible to deduce the exact fields returned, although we know the queries. Nevertheless, it was possible to roughly describe the traffic and to assess the cache efficiency and the usage of the tape backend.

For ECFS, we saw read accesses on only 9% of the file corpus with a disk cache hit ratio of 86.7% during the observed timespan. Only 26,3% of the files on tape were ever read. The MARS logs do not reveal deeper information about which files and fields were accessed, but show that 95.1% of the requested fields were served from disk caches where only 2.2% of all requests included accesses to one or multiple tapes. Despite of this strong caching infrastructure, we have observed more than 9.5 million tape loads over a period of two years with 5% of the tapes being loaded more than 1,000 times.

The archival system's latency is not the primary bottleneck for the computations, as most operations run in batch and wait until the requested data is available. Nevertheless, since extensive queries that involve tapes can take several minutes, hours or even days, it is worthwhile to improve the average loading time of the tapes. Although the system is already well-configured, the 60 second tape reload rate of 14.8% and the prefetching analysis expose further potential for optimization.

The quality of weather forecasts constantly improves due to faster computers and improved computational models. At the same time, the requirements for storage infrastructure grow as well. Kryder's Law states that the areal density of bits on disk platters roughly doubles every two years [Wal05]. While this was true for the last three decades, Rosenthal et al. forecast that the improvements in storage cost per bit for disk and tape will be much slower [RRM+12]. ECMWF faced a CAGR of 45% over the last years, which lately

increased to 53% with the latest supercomputer. While this growth was maintainable with a constant budget during the last years, it might lead to the economical threats for long-term digital storage described by Baker et al. [BSR+06]. The consequence has to be an adjustment of scenario planning or implementing means to grow slower.

Due to a lack of studies, it is difficult to compare or generalize our results to other archival storage environments. Nevertheless, we believe that they are also relevant for other existing or future systems and that they can help to make the right design decisions. The reason is the observation that many large systems share at least some essential properties like the small read-to-write ratio and the overall architecture combining a large tape library with a relatively small disk-based cache. We developed a generic, extensible set of tools that can be applied to analyze workloads of archives and data centers. The cache simulation, for instance, helps to evaluate new caching strategies and to explore the impact of different eviction strategies and cache bucket sizes.

2.9 Closing Remarks

We are very grateful to the European Centre for Medium-Range Weather Forecasts for the opportunity to browse and analyze their log files and to share their valuable knowledge on building large scale archival systems.

The analyzed traces are stored at ECMWF and will be made available upon individual requests. The cache simulator, part of the scripts, and links to the trace files are available at https://github.com/zdvresearch/fast15-paper-extras.

3 | Evaluation of Applied Intra-Disk Redundancy Schemes to Improve Single Disk Reliability

Abstract

Exponentially growing capacities of disk drives have increased the problem that not only a complete disk can fail, but also individual, small groups of sectors can be erroneous. These sector errors are especially critical during RAID rebuilds because they can only be detected when the corresponding sectors are read. Mechanisms to cope with sector errors, therefore, have become an important way to improve disk reliability. One approach to deal with sector errors is the introduction of intra-disk redundancy, where additional redundancy blocks are calculated and stored for each set of disk sectors.

Previous studies have introduced intra-disk redundancy schemes and have evaluated their impact on disk reliability. None of these studies has evaluated the influence on disk drive performance or the underlying energy consumption. The study presented in this paper benchmarks existing schemes concerning these metrics. It shows the surprising result that weaker codes combined with newly introduced scrambling techniques can produce faster layouts with similar reliability properties than previously proposed strong codes.

3.1 Introduction

Magnetic disk drives are still the most commonly applied devices to store data, and they are used in many environments where data losses cannot be tolerated, like in file servers, archives, or backup systems. Next to complete disk failures, additional error types like sectors that can no longer be read by the disk, called latent sector errors (LSE), or reads that deliver wrong results, called block corruptions, have to be considered.

The probability of latent sector errors and corrupt blocks has not increased per access over the last years. Nevertheless, the increasing bit and track densities of today's hard disk drives and their resulting exponentially growing capacities have led to an increase in the amount of stored data that makes this small probability very relevant. Studies on disk failures revealed that, next to full disk failures, LSEs and corrupt blocks are a big threat to data reliability [BADAD+08]. They are especially difficult to handle because they are only detected when corresponding sectors are read.

Latest studies from Schroeder et al. [SDG10] present a new statistical failure model based on findings of Bairavasundaram et al. [BADAD+08] and evaluate different intra-disk redundancy (IDR) schemes concerning their reliability. While the reliability of the known IDR schemes has been evaluated, neither a performance evaluation nor the impact of applied codes on the overall energy consumption has been analyzed yet.

The motivation behind this work is to sharpen the understanding of the compute and storage overheads of intra-disk redundancy to improve the reliability of a single hard disk drive. What are the challenges to implement IDR codes into the storage stack, and what is the impact on performance and energy consumption? To analyze this, we have implemented an iSCSI target [SMS+04] that transparently incorporates multiple mechanisms like IDR-codes, caching, interleaving and check-on-read to increase single disk reliability and a test environment that enables us to exactly measure the power consumption of the system under test. We conducted tests with various workloads and parameterizations to carve out the impact and benefits of different strategies.

A major contribution of this paper is a discussion and experimental evaluation of intra-disk redundancy. We also introduce a new general scheme how to improve the reliability of cheap codes. It is important to notice that we do not present new IDR codes but evaluate existing analytical results on their usability and derive new conclusions from these findings.

Our results show that applying IDR codes on disk drives requires a careful parameterization and that no code fits all requirements. While sequential accesses, as found in archive setups, merely suffer from IDR encoding, small random workloads strongly degrade performance. Using stronger codes also results in a stronger degradation of performance due to a raised parity group size, leading to a higher number of read-modify-write cycles and a raised compute and storage overhead. To overcome these shortcomings, we propose to reorder the blocks of contiguous parity groups to maximize the physical distance of logically contiguous blocks. We call this technique *scrambling* and show that modern disks allow

us to apply these techniques with a considerably small performance penalty. By applying scrambling to intra-disk-redundancy, even weak codes can compensate error bursts that would otherwise require much stronger and more expensive codes.

3.2 Related Work

Studies about huge disk populations [SG07, PWB07, BADAD+08] and their evaluation by Schroeder et al. [SDG10] led to well understood disk drive failure models that include latent sector errors and corrupt blocks as huge threats.

To improve the reliability of disk-based storage systems, multiple mechanisms could be applied. Typical ways to cope with these classes of errors are RAID schemes [PGK88] to compensate failing disks and to recover from detected bad blocks and disk scrubbing to proactively find corrupt data [SXM+04, IHHE08]. Other ways are file systems that integrate multiple mechanisms, like the IRON file system [PBA+05], or applying intra-disk redundancy codes that use RAID-like schemes on a single disk to recover from multiple bad blocks on a disk drive [WT07, DEH+08, SDG10, PBA+05]. The focus of this paper is on IDR-codes.

Dholakia et al. derived expressions for the MTTDL of RAID 5 and RAID 6 in the presence of latent sector errors and disk failures [DEH+08]. They proposed the interleaved-parity-check (IPC) scheme as an alternative to single-parity-check (SPC) [DEH+08]. They conceptually arrange data sectors in a matrix and calculate one parity block per column, which are also called interleaves. A segment consisting of m columns tolerates error bursts of length up to m but only one error per column. Dholakia et al. claim that IPC can achieve the same levels of reliability as Reed-Solomon (RS) encoding. A later analysis [IHHE08] concludes that IPC can lead to the same level of reliability that can be found in a hypothetical system lacking sector errors.

Based on the statistical error models and the analysis of [BADAD+08], Schroeder et al. analyzed five intra-disk redundancy schemes and different disk scrubbing policies concerning their reliability. Interestingly, they showed that IPC provides considerably worse reliability expectations in their setting than RS-codes. The authors blame the strong correlation of LSEs and long-tail behavior of error burst lengths for these results. Consequently, they propose to employ the well-known row diagonal parity scheme in intra-disk redundancy and call their variation column diagonal parity (CDP). Nevertheless, the study does not evaluate the practical feasibility of CDP and the performance impact on the storage stack.

Other codes that were developed for RAID schemes, failure resilient communication, or IDR and could be applied to improve storage reliability include Tornado [WT07], WEAVER codes [Haf05], Reed-Solomon [RS60], EVENODD [BBBM95], RDP [CEG+04], or liberation codes [Pla08b]. Plank et al. have conducted a performance evaluation of open-source erasure coding libraries [PLS+09].

Next to the bare performance of storage systems, the energy consumption of storage systems has emerged as a critical factor. Zedlewski et al. measured hard disk power consumption to integrate an energy consumption model into the DiskSim simulator [ZSG+03]. They measured two different disks that were interconnected via PCMCIA. The testing environment introduced a shunt resistor into the power circuit, and its voltage was measured by a digital multimeter. Chen et al. [CGK+10] used a similar setup to measure the disk drives power consumption. However, they introduced a separate laboratory power supply for the disk and have been able to measure up to 50K power samples per second. They evaluated the opportunities of disk drive acoustic modes for power management, and it has been shown that these modes do not help to save energy. Sehgal et al. benchmarked the power consumption of multiple file systems under multiple workloads [STZ10].

The best performance for a disk drive can be achieved if the physical conditions and capabilities can be exploited as far as possible. Gim et al. have shown that the performance of disk accesses can be significantly improved for known hard disk drive characterizations [GWC+08]. Effects of a possible track alignment have been investigated in other contexts, e.g., by Schindler et al. [SGLG02], Qian et al. [QMW08], or Lensing et al. [LMB10]. These papers motivate us to match an on-disk format to the disk drive geometry.

3.3 Challenges of applying IDR codes

Intra-disk redundancy enriches data on a hard drive with additional redundancy information that has to be stored on the same device. This redundancy information reduces, comparable to classical RAID schemes, the usable disk capacity by a factor $r = \left(1 - \frac{m}{m+k}\right)$, where m defines the number of data blocks in each on-disk parity group (PG) and k defines the number of redundancy blocks for each parity group. In this paper, the term parity group is used as the set of data blocks for which a set of redundancy blocks is calculated for. The size of each block b influences both the size of the logical PG and its IDR-encoded representation (ePG) that is written to the physical disk. Therefore, a PG consists of m data blocks, has a logical size of $m \cdot b$, and an ePG size of $(m + k) \cdot b$.

On read accesses, only the data blocks of a PG have to be read. If the disk detects a bad sector by itself, the recovery mechanisms of the applied IDR codes could be used to restore corrupt sectors. In this case, the full ePG is read, and the block is restored and rewritten to disk - relying on the remapping capabilities of the disk drive.

An additional threat to data integrity are *corrupt blocks*, which are not detected by the disk and lead to erroneous data inside the accessing applications. To cope with this class of errors, we evaluated a mechanism that fetches the full corresponding ePG of a requested block, re-encodes the PG, and compares the redundancy blocks. If they do not match, at least one block contains erroneous data and needs to be recovered. This results in a compute intensive task as the corrupt block has to be identified and recovered. The mechanism is called *check on read*, and its overhead is evaluated in Section 3.5.6.

To update a single data block of a parity group, at least some additional data and redundancy blocks have to be read, and the new data block and at least some redundancy blocks have to be written. Therefore, a single updated block in a parity group might lead to an expensive read modify write (RMW) cycle. Hence, the best performance is achieved for large writes as no information must be read and the full parity group could be written to disk at once.

Caching mechanisms that use read-ahead and write-back mechanisms can be integrated into the block device to reduce the number of RMW cycles. Assuming request locality, like for sequential writes, a parity group that is only partially updated should not immediately be written to disk but kept in cache until either all of its data blocks are updated or a timeout occurs. If the block device is used in the O_SYNC mode, these caching mechanisms cannot be applied. All parity groups that are (partially) updated by a write request have to be re-encoded and flushed to disk to fulfill the O_SYNC semantics.

The size of a parity group is a critical factor. For small values of m, k, and b, the parity groups will also be small, which leads to a reduced probability of RMW-cycles but also to a small size of the ePG, which in turn leads to a higher impact of error bursts. Schroeder et al. [SDG10] have shown that latent sector errors exhibit a locality and that one LSE raises the probability of additional LSEs within a distance of 10 sectors of the error. To enhance disk drive reliability, either short parity groups should be spread to lessen the impact of error bursts, the IDR-codes should be parameterized with a high number of redundancy blocks, or b should span multiple disk sectors so that consecutive sector errors only hit few logical blocks. In this paper, we explore all these variants by implementing scrambling routines, evaluating codes with a high reliability, and using different block sizes for IDR codes.

3.4 Parity Group Scrambling

Schroeder et al. have shown that a latent sector error increases the probability of additional errors in its neighborhood [SDG10]. This implies that the different blocks of a parity group should be separated far enough to cope with these bursts. However, separating consecutive blocks by too many sectors leads to performance degradations as the mechanical components of the disk limit the number of possible random I/Os per second [SSP+05].

Achieving both high performance and high data reliability requires careful handling of the data layout and a closer look at the error distribution. If an LSE occurs, less than 2.5% of the errors consisted of two erroneous sectors and less than 2.5% consisted of a burst of more than two errors. Furthermore, not only the burst length but also spatial locality has a strong impact on data reliability. Schroeder et al. showed that 20%-60% of all errors have a neighbor within a distance of less than 10 sectors in logical sector space. Furthermore, they speculate about bumps with raised error probabilities for adjacent sectors [SSP+05]. These findings lead to the assumption that spreading ePGs yields an improved reliability because logically consecutive blocks will not be physically located near each other.

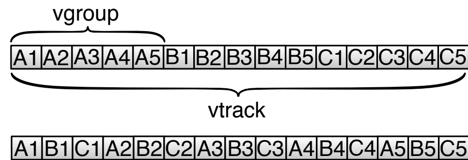


Figure 3.1: Vtrack shifting for $g = 5$ and $s = 3$.

Modern disks can access multiple consecutive tracks without moving their head and full tracks are read or written. Furthermore, huge disk caches allow us to buffer, schedule, and optimize read and write requests on disks. Due to disk drive heterogeneity and the lack of standardized mechanisms to describe or determine the correct geometry information, we use virtual tracks (vtrack) as an abstraction of approximated physical disk tracks.

We propose to reorder the blocks within a vtrack so that correlated blocks of a PG are pulled apart to maximize their physical distance to each other to reduce the impact of error bursts. A vtrack consists of s vgroups that each contain g logically contiguous blocks of size b. The vtracks add another layer of abstraction so that logical block addresses on top of the vtracks need to be mapped to their actual physical address. As the vtrack's size is fixed, a given logical block address can be mapped to its corresponding vtrack. The parameter s determines the shift of the block. Figure 3.1 shows an example of a vtrack that consists of $s = 3$ vgroups that contain $g = 5$ blocks. A huge s results in a huge physical distance of the blocks.

In contrast to other work, we do not propose a new IDR-scheme but provide an abstraction of an on-disk format that holds as a container for any IDR-code. This container can provide nearly arbitrary protection against bursts of sector errors because the blocks belonging to an ePG can be separated arbitrarily far from each other. While interleaving of consecutive blocks to improve the performance was already used in early disk driver designs and the spreading of consecutive blocks was also used in other IDR codes, our proposed abstraction is a new and general solution to improve the reliability of a disk drive.

3.5 Evaluation

We have built an automatized and distributed framework to measure the impact of both throughput and related energy consumption. Our test environment consists of an Intel Atom D510, a 64-bit 1.66 GHz dual-core CPU with 2 GB DDR2 RAM, and an internal 80 GB WD800JB hard disk, which stores the operating system and the application programs. For our measurements we used two Western Digital 2 TB 3,5" disk drives with 64 MB cache and 4 kB native sector size (WD20EARS) and 32 MB cache and 512 byte sector size (WD20EADS). The disks represent the state of the art for energy efficient mass storage

and feature two different native sector sizes. We chose the Atom CPU as it is designed for energy efficient systems and can be integrated into embedded systems. All machines in our test bed are interconnected via Gigabit Ethernet.

We implemented a modular iSCSI target using the 2.0-rc of the SCST framework[1] (SVN revision 1971) [SMS+04]. This framework was chosen as it allows us to implement a virtual block device in user space and to export it via iSCSI to test it in a network setup and to use it as a local block device. We have implemented a *tracer* module that exports an existing block device and logs incoming read and write requests alongside their parameters. The *scrambler* module implements the mechanisms presented in Section 3.4. The *coder* module transparently incorporates multiple intra-disk redundancy codes. An overview of the system is given in Figure 3.2.

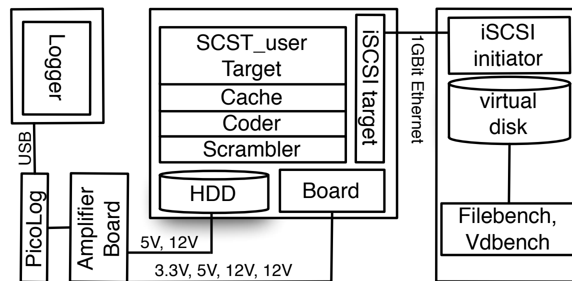


Figure 3.2: Overview of our evaluation setup.

The test system runs a 64-bit Ubuntu 10.04 Server (2.6.32-15 kernel) and applied SCST-patches. For IDR coding, we use the zfec[2] libraries for Reed-Solomon codes and our implementations of IPC, SPC, and CDP. We implemented these codes within our framework and created unit tests to check their correctness. The target can be configured to use the scrambling routines and a parameterized IDR code. It includes a switch to execute local writes with *O_SYNC* and a switch for check-on-read mechanisms for read requests. All these mechanisms are transparently incorporated into the block device and are computed on the test system.

To separate the test logic and benchmarking overhead from the test system, we added a dedicated client machine that mounted the exported block device via iSCSI. This client automatizes the benchmarking tools FileBench 1.4.8.fsl.0.7[3] and Vdbench 5.02[4]. Filebench was used to generate macro benchmarks to compare workloads on top of a file system. Vdbench was used for micro benchmarking to generate individual workloads on a block device level to carve out specific block device behaviours.

[1] http://scst.sourceforge.net/
[2] http://pypi.python.org/pypi/zfec
[3] http://www.fsl.cs.sunysb.edu/˜vass/filebench/
[4] http://sourceforge.net/projects/vdbench/

For the energy measurements, we used a Picolog 1216 data acquisition unit[5] with a modified driver that allows measurements to be set up, triggered, and collected by our distributed test framework. The logger collects data from a newly developed circuit that allows us to simultaneously monitor the 12 V and 5 V power supply of the hard disk under test, the 3.3 V, 5 V, and 12 V power supply of the ATX-connector and the 12 V CPU-connector on the motherboard. The current is measured by inserting a shunt resistor into the supply rails. The voltage drop across a 10 mΩ shunt is small enough to comply to the ATX specification as long as the current is not bigger than 15 A. This small voltage difference is amplified by our circuit before being forwarded to the data acquisition unit. Furthermore, the difference voltages are referenced to ground, so that they can be measured by the same data acquisition unit without problems induced by potential differences.

The logger unit is able to measure each channel every $300\mu s$, which results in a total of 3.3k samples per second per channel. Given the 12 bit resolution for the 2.5 V range of the Picolog 1216 and the signal amplification factor of our circuit (64 for 3.3 V and 5 V and 32 for 12 V), the smallest measurable resolutions are 3.15 mW for 3.3 V, 4.77 mW for 5 V, and 22.89 mW for the 12 V lines.

3.5.1 Power Consumption

We measured the target system in defined states to be able to analyze the relative differences in the overall energy consumption. We executed all tests multiple times to fit the results into a confidence interval of 95%.

Idle The system is booted and left idle for a long time.

Seek We wrote a tool that opens the hard disk with the `O_DIRECT` flag and executes small direct reads and writes on alternating sides of the disk so that the disk's head moves as much as possible.

Seq Zeros are sequentially written to disk by the command
`dd if=/dev/zero of=/dev/sdb bs=512k`

CPU Runs the `cpuburn` utility that brings all cores to 100% load. The hard disk was sent to sleep with `hdparm -y`.

Rand Vdbench run with `O_DIRECT` 4 kB writes randomly distributed over the full disk.

The results of Figure 3.3 present the summarized 5 V and 12 V lines of the HDD under test and the summarized values for the 3 ATX power supply lines and the 12 V CPU connector of the mainboard. The most power on the HDD line is required for large sequential writes, followed by a workload that mostly consists of large seeks, followed by the random workload. Interestingly, the WD20EARS (R), while having twice the cache

[5] http://www.picotech.com/multi-channel-daq.html

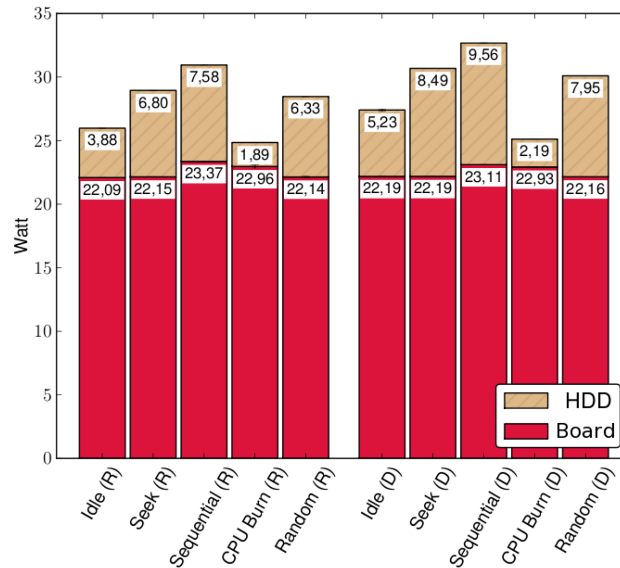


Figure 3.3: Energy consumption of HDD and Board (Mainboard + CPU) for defined system states for WD20EARS (R) and WD20EADS (D) running on an Intel D510.

and a better overall performance than the WD 20EADS (D), consumes less power for all workloads.

While the differences of the HDD power consumption are obvious, we were not able to measure noteworthy differences on the ATX-lines. For the Idle and CPU workloads, the difference is less than 1 Watt, and the most power is required for the sequential workload. The measurements show that the 12 V CPU power supply line is not used to power the CPU as only very small voltages were measured. This shows that the CPU does not support any speed stepping and always consumes a nearly constant amount of energy, while the chipset requires more energy for workloads like the sequential read.

3.5.2 iSCSI Access Patterns

To sharpen our understanding of the distribution and sizes of block level requests, we ran multiple benchmarks and traced all incoming iSCSI target commands. We have exported a target block size of 4 kB so that all multiples of 4 kB up to 512 kB might occur as request sizes and mounted this block device on the dedicated test client.

We created block level accesses with the `dd` utility and some Vdbench configurations for sequential and random workloads with varying block sizes. Furthermore, we investigated file system based traces by creating an ext4 on the exported block device and running the `cp` utility and some Filebench personalities like OLTP, Fileserver, and Videoserver.

In summary, the results show that for sequential workloads, like cp, dd or the video server, write requests of 512 kB dominate the workload, while workloads with randomized traffic are dominated by 4 kB requests. Next to the write requests, where sequential

accesses appear mostly with a size of 512kB, the typical size for sequential reads is only 128 kB. A read request size bigger than 128 kB occurred in none of our tests. These findings were used to reduce the parameter space for the evaluation in Section 3.5.4.

3.5.3 Intra-Disk Redundancy Codes

We conducted multiple encoding tests to evaluate the reasonability of code parameterizations and to detect bottlenecks in the I/O stack of our target hardware. To test the theoretical assumptions of Schroeder et al. [SDG10], we chose Single Parity Check codes. They add a single XORed parity block to a set of data blocks in two configurations, *SPC 8+1* and *SPC 16+1*. Additionally, we implemented IPC codes proposed by Dholakia et al. [DEH+08] and evaluated the parameterizations *IPC 8+2*, *IPC 16+2*, and *IPC 16+4*. They are able to compensate any error bursts of lengths 2 and 4 and can recover up to 2 and 4 defect blocks for some erasure patterns. While Schroeder et al. also evaluated *CDP*, we omit them in this subsection due to their configuration inflexibility. They cannot be configured to be comparable to the other codes, but they will be analyzed in Section 3.5.5. Reed-Solomon (RS) codes belong to the most robust codes and are able to recover any k faulty blocks for an $(m + k)$ encoded parity group with m data and k parity blocks and are used as a reference for the reliability analyses. We evaluated *RS 8+2*, *RS 16+2*, and *RS 16+4*. The resulting on-disk formats and the corresponding redundancy groups (blocks with same color) for these codes are shown in Figure 3.4.

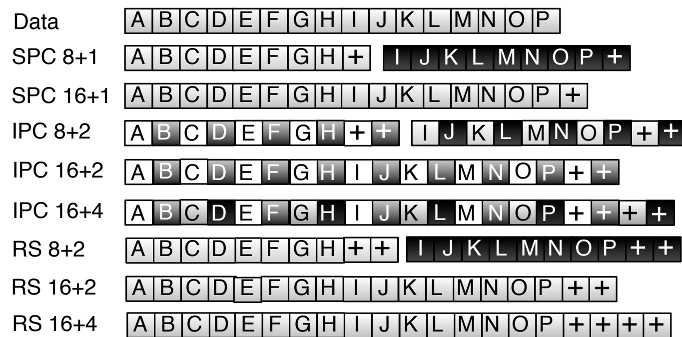


Figure 3.4: Redundancy groups for on disk formats.

To measure code performance, we created a RAM disk containing a 1 GB file, which was encoded using the abovementioned codes, and the results were written to `/dev/null`. The tests were executed with 1,2,4, and 8 encoding threads in parallel. Figure 3.5 presents the results where the three bars for each configuration represent the used block sizes for the algorithm (from left to right: 512 B, 4 kB, 64 kB). All tests were repeated multiple times and are in a 95% confidence interval.

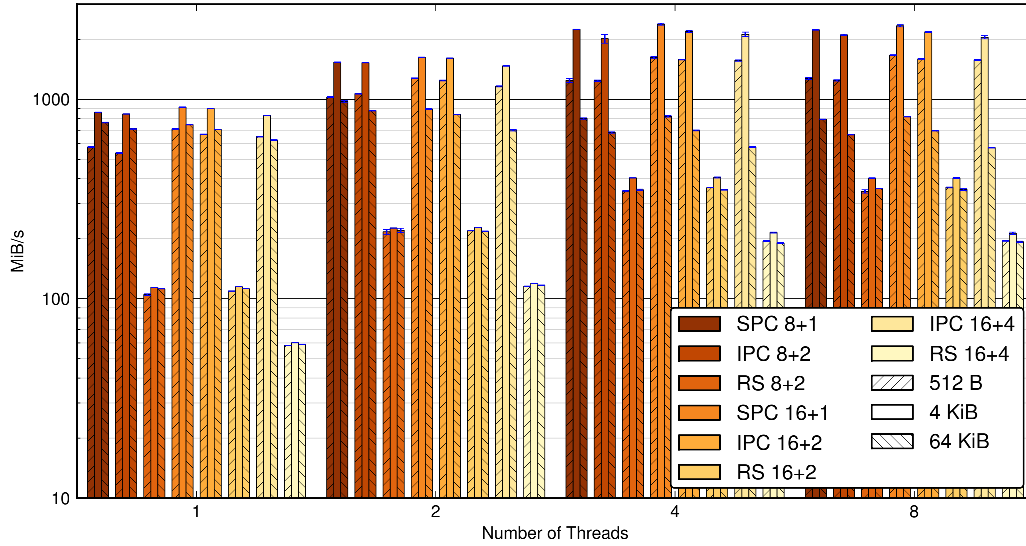


Figure 3.5: Throughput analysis for in-memory encoding on Intel Atom D510 in logarithmic scale.

The results show that a block size of 4 kB always outperforms the other block sizes and that the CPU can fully utilize all of its 4 cores to speed up a parallel encoding process. Another observation is the huge performance difference of the SPC and IPC codes that merely consist of XOR operations compared to the RS codes. Given the encoding results for RS codes and an approximated disk throughput of $120\frac{MB}{s}$, a single disk with applied RS coding for IDR would fully saturate the CPU under test. This again motivates us to find alternatives to RS codes for IDR with similar reliability guarantees.

3.5.4 Full I/O stack

This section evaluates the throughput and energy consumption of our iSCSI target implementation with enabled scrambling and applied IDR codes for multiple workloads. We used a dedicated client to run the tests via a mounted iSCSI target so that the system under measurement is only used for block device related operations like IDR coding, scrambling, and disk accesses. The hosts were connected with 1Gbit. One motivation for this evaluation is to better understand how we should parameterize IDR codes on disk, especially as it has been shown by Schroeder et al. that this parameterization has a strong impact on reliability [SDG10]. One of their findings was that a $16 + 2$ RS-code has similar reliability properties to a $56 + 7$ IPC code and that higher order RS codes outperform corresponding IPC codes by an order of magnitude. The open question is which of these codes is more practical in real environments? Might it be more beneficial to spread data over the disk than to calculate more sophisticated codes?

For writes, every updated block results in a (partial) re-encoding of a parity group and a write request to disk. To alleviate the number and impact of these RMW-cycles, we

implemented a write-cache that merges consecutive writes and delays the encoding and disk access until the full parity group is written or the cache is flushed due to a timeout or eviction. This cache improves the overall throughput but cannot fulfill the semantics of a disk mounted with `O_SYNC` or `O_DIRECT`. We ran several tests with synchronous writes and disabled caching. With this configuration all, write requests result in RMW cycles and re-encodings of parity groups, which renders synchronous operations with applied IDR as impractical. Therefore, we canceled further investigations of synchronous operations.

To keep the parameter space manageable, we restricted the tests to three different parameters for the scrambler ($s = 0, 16, 128$), where $s = 0$ means no scrambling, and the same block sizes ($b = 512$ byte, $4\,\text{kB}$, $64\,\text{kB}$) and IDR parameterizations as examined in Section 3.5.3.

We used Vdbench for random and sequential block level accesses and exported the full 2 TB disk via iSCSI. Furthermore, we used Filebench to evaluate a file system based access with the *fileserver* and *videoserver* personalities on top of an xfs file system. As the creation of the file system before each test is very time intensive, we decided to use xfs for its fast file system initialization capabilities. We also reduced the exported block device size to the first 40 GB of the disk for the Filebench test runs.

Before each test, the target system was rebooted, the iSCSI initiator was connected, (for the Filebench tests) formatted with xfs and mounted. Then, the benchmarks were started on the client machine, and after an initial phase of 60 seconds, 60 seconds of data were captured. According to our measurements of Section 3.5.2, the fileserver workload mainly consists of write requests ranging from 4 kB to 256 kB, and the video server mainly consists of 512 kB writes.

The graphs in Figures 3.6 and 3.7 present the disk's throughput (bottom) confronted with the corresponding energy consumption (top) for the abovementioned configurations. The Vdbench tests were conducted with request sizes 4 kB and 512 kB for writes and 4 kB and 128 kB for reads. These parameters were chosen to represent the findings of the iSCSI access pattern analysis in Section 3.5.2. Each set of eight bars, which represents the eight code parameterizations, is shown for a combination of the scrambling factor $s = 0/16/128$ and the IDR code block size $b = 512$ byte$/4\,\text{kB}/64\,\text{kB}$. The error bars represent a 95% confidence interval.

The results clearly show that strong codes and scrambling can be used for sequential workloads but suffer huge performance drops for random workloads. This can be explained by the internals of the target. Our implementation adjusts its scrambling routines to match the parameter g to the number of blocks of an ePG. A $16 + 4$ code ($g = 20$) with a block size of $64\,\text{kB}$ and $s = 128$ would result in an ePG size of 1280 kB and a vtrack size of 160 MB. Our caching assembles multiple writes to be encoded and flushed so that a sequential write workload would be cached. In the best case, a chunk of 1280 kB could be written to disk at once. The worst throughput was measured for randomly distributed 4 kB writes as they result in encoding and writing full PGs.

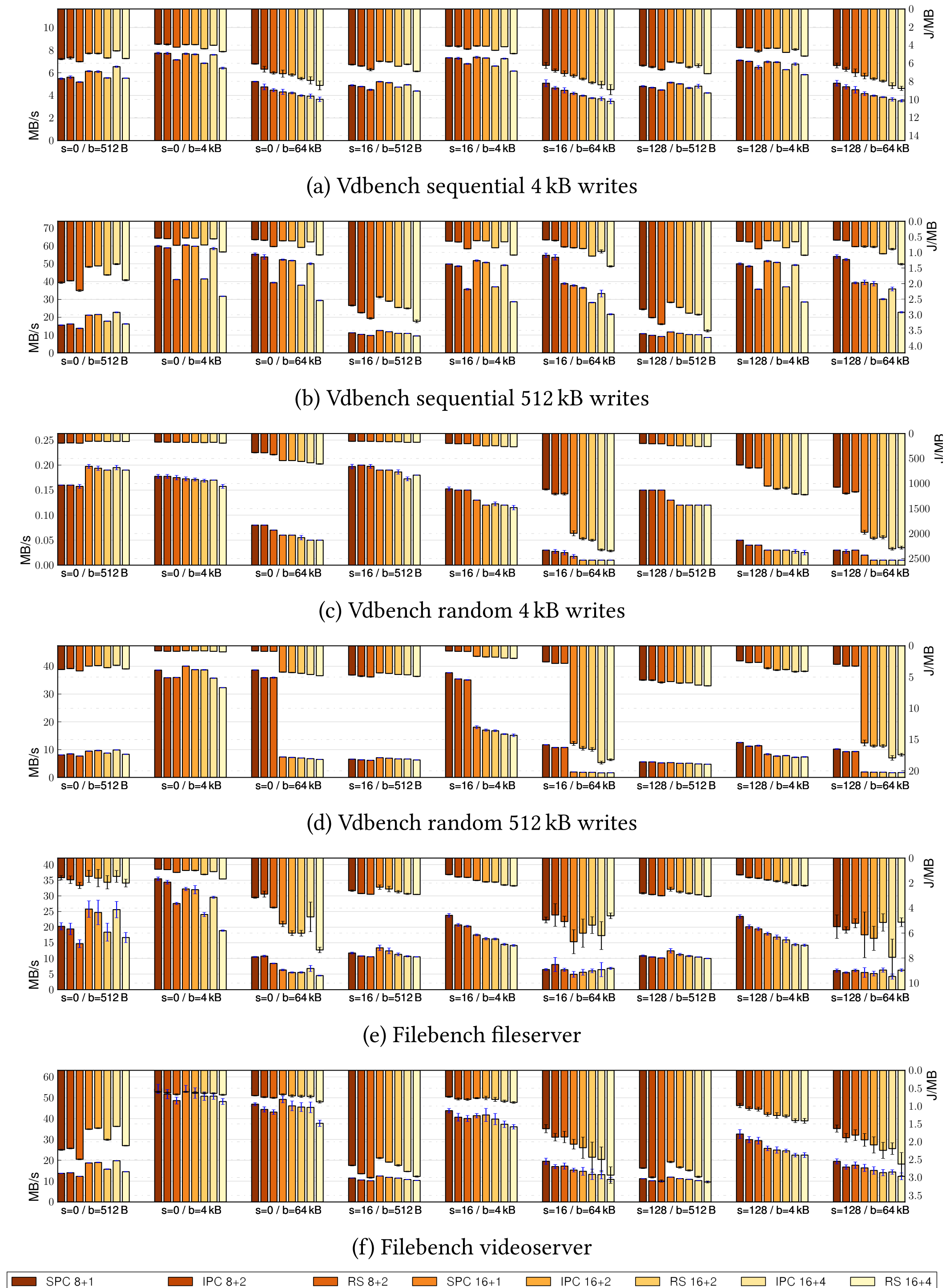
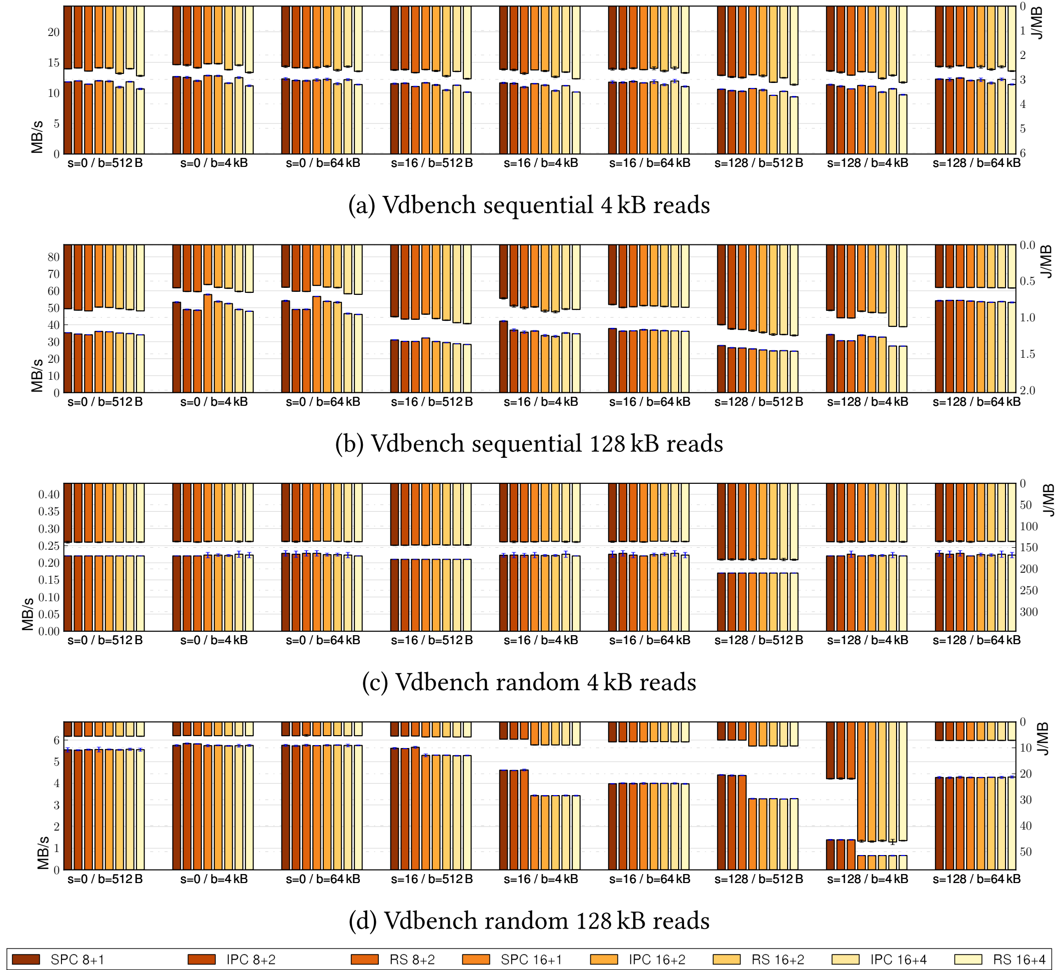


Figure 3.6: Throughput (bottom) and disk energy consumption (top) analysis of full I/O stack for WD20EADS



(a) Vdbench sequential 4 kB reads

(b) Vdbench sequential 128 kB reads

(c) Vdbench random 4 kB reads

(d) Vdbench random 128 kB reads

Figure 3.7: Throughput (bottom) and disk energy consumption (top) analysis of full I/O stack for WD20EADS

While the sequential workloads merely suffer from scrambling and large block sizes, the random writes can only hold the throughput for 512 kB writes with codes $(8 + x)$, $s = 4$, and 4 kB block size.

Our read implementation uses asynchronous `LIO_LISTIO` calls to fetch all corresponding data blocks for a read request. We do not provide an additional caching for reads but rely on the operating systems page cache. For sequential 4 kB read requests, the throughput is nearly constant while 128 kB reads suffer from the more randomized disk access patterns induced by the scrambling.

The fileserver workload in Figure 3.6e with $s = 0/4$ kB shows the impact of random writes with different sizes. Due to many partially updated parity groups, the strong codes yield a stronger performance drop as for the sequential videoserver workload. This also holds for the impact of scrambling that still yields an acceptable performance for the

videoserver workload ($s = 16/4$ kB). For all write workloads, the throughput in general suffers from a block size of 512 byte compared to the 4 kB configurations.

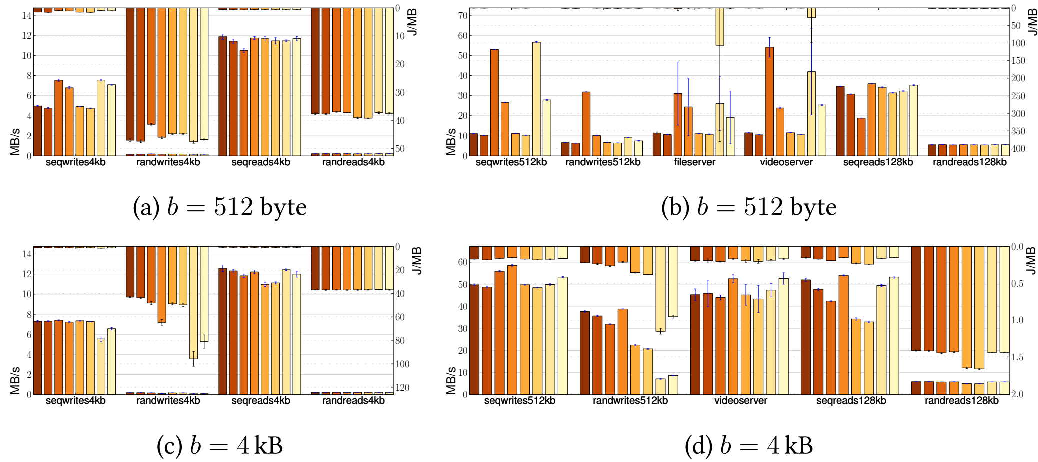
We ran all tests of this section on both disks but only presented the results for the WD20EADS in detail as the results for the WD20EARS were very similar. The presented disk also has 512 Byte native sector size, which corresponds with the analyzed disk population of the findings this paper relies on [SDG10, BADAD$^+$08]. By summing up all throughput rates of all tests, we found that the WD20EADS was on average $2.3\frac{MB}{s}$ faster for $b = 512$ byte and $3.3\frac{MB}{s}$ faster for $b =$4 kB but on average $9.1\frac{MB}{s}$ slower for $b = 64$ kB block sizes.

3.5.5 Reliability comparison

We have evaluated multiple IDR codes concerning their throughput and energy consumption and have shown that the stronger codes have a negative impact on these properties, which makes them impractical to apply. In this subsection, we analyze weak codes like SPC $8+1$ or IPC $8+2$ with scrambling. We think that they have similar reliability properties like stronger codes regarding the findings of [SDG10]. Due to the unavailability of the trace data used by Schroeder et al., it is impossible to argue about the expected reliability of the codes for arbitrary settings. Instead, we focus on the length of error bursts, which can be tolerated by the codes, and analyze different codes for the amount and types of errors they can compensate alongside their throughput and energy consumption.
We benchmarked additional configurations of the codes introduced in Section 3.5.3 and our implementation of CDP. We used the WD20EADS disk drive in the same environment as described in Section 3.5.4 and present the results for IDR code block size $b = 512$ byte and $b = 4$ kB in Figure 3.8. To be able to compensate a single error burst of length up to 4, we propose configuration *A*. To compensate two bursts of length 4 and some other combinations of errors, we evaluated the codes *B*,*C*, and *D*. The same properties hold for the codes *E-H*, but their parameterizations allow us to compensate error bursts of length up to 10 (see Table 3.1).

ID	Code	$m+k$	s	1-r	ePG size b=512, 4 kB
A	SPC	8+1	4	89%	4,5 kB, 36 kB
B	IPC	8+2	4	80%	5 kB, 40 kB
C	CDP	16+8	-	68%	12 kB, 96 kB
D	IPC	32+4	-	89%	18 kB, 144 kB
E	SPC	8+1	10	89%	4,5 kB, 36 kB
F	IPC	8+2	10	80%	5 kB, 40 kB
G	CDP	100+20	-	82%	60kB, 480 kB
H	IPC	80+10	-	89%	45 kB, 360 kB

Table 3.1: Codes under test with parameterization $m + k$, scrambling s, resulting storage efficiency $1 - r$ and resulting ePG sizes for IDR code block sizes 512 byte and 4 kB.



(a) $b = 512$ byte

(b) $b = 512$ byte

(c) $b = 4\,\text{kB}$

(d) $b = 4\,\text{kB}$

Figure 3.8: Throughput (bottom) and disk energy consumption (top) analysis of IDR codes defined in Table 3.1 for WD20EADS. The bars represent the codes *A*-*H* (from left to right).

The results of Figure 3.8 show that IDR codes heavily depend on the sizes of the resulting requests to the disk drive. For $b = 512$ byte (see Figures 3.8a and 3.8b), especially the CDP code shows the best throughput for writes but also the worst for reads due to its high storage overhead. Code *C* results in 96 kB blocks that are written to disk at once, while code *A* results in write requests of 4,5 kB. These findings make the case for stronger codes resulting in bigger parity groups or a bigger IDR code block size. Therefore, we evaluated $b = 4\,\text{kB}$ for codes *A*-*H* in Figures 3.8c and 3.8d. These results show that the weaker codes *A* and *B* with $s = 4$ can cope with the performance of the strong codes. On average, they even result in a better disk energy consumption. The results of Figure 3.8d for random 512kB writes especially address the problem of (partially) updated parity groups that need to be re-encoded and written to disk. A single write may result in the re-encoding and writing of multiple full parity groups for the strong codes.
Depending on the native sector size of the disk and the block size of the applied codes, the single disk reliability could even be improved. Given 512 byte sectors and 4 kB IDR code block size, every error burst of up to 9, and some of up to 16, contiguous sectors can be recovered with an $x + 2$ IDR code as each IDR block covers 8 native disk sectors. Therefore, a bigger IDR code block size improves disk reliability but at the price of bigger parity groups and the related performance degradation. To achieve the same improvement for disks with 4 kB native sector sizes, like the evaluated WD20EARS, the IDR codes have to be parameterized with a block size of 64 kB, which resulted in a lower throughput than the 4 kB block size (see Figure 3.6 and Figure 3.7).

3.5.6 Check on Read Overhead

Section 3.3 introduced our simple approach of a check on read mechanism. For each read block, the corresponding ePG is retrieved from disk and the PG is fully re-encoded and compared to the ePG's redundancy blocks. Due to the sequentiality of the ePG's on-disk format and the performance of the IDR encoder, we consider this simple mechanism as a possible implementation to cope with corrupt blocks. To evaluate the induced overhead, we executed the sequential and random read Vdbench tests as described in Section 3.5.4. The induced overhead varies with the parameterization of the I/O stack. The penalty for check on read is negligibly small without scrambling and for block sizes of 512 byte and 4 kB. The block size has the biggest impact on performance, especially if the sizes of the coding blocks are bigger than the sizes of I/O requests. Even without scrambling, factors of 2 for random reads and factors of 5 and worse for sequential reads occurred in our measurements. The biggest overheads have been observed when using a block size of 64 kB in conjunction with scrambling factors of 16 and 128. This result was expected because for each read request considerable amounts of data have to be retrieved and verified. However, for small coding block sizes of 512 byte and 4 kB and scrambling factors of up to 16, check on read provides a viable method for ensuring data integrity (detection of corrupt blocks). At the same time, it does not sacrifice additional disk space and does not induce additional management overhead for storing digests.

3.6 Conclusion and Discussion

This paper gave an overview of the challenges of improving single-disk reliability with intra-disk-redundancy. We implemented a virtual block device and benchmarked the performance and energy impact of applied IDR codes alongside our proposed scrambling mechanisms for various workloads. The investigation of IPC, SPC, CDP, and RS codes with different levels of redundancy and block sizes revealed differences in their computational throughput and in their applicability in the storage stack. Our evaluation showed that there is always a tradeoff between performance degradation and gained extra reliability, which heavily depends on the workloads and parameterization of the codes.

Strong IDR codes are impractical for (synchronous) random small writes (like oltp) because parity groups are only updated partially. This results in expensive read-modify-write cycles that reduce the throughput and raise the energy consumption. On the contrary for large sequential writes, IDR codes can be applied nearly not affecting the performance because full parity groups can be encoded and written to disk at once. To overcome the inflexibility of huge parity groups we proposed to reorder blocks over multiple parity groups. This enables the storage system to compensate error bursts even with weak codes. Our evaluation showed that there is no parameterization that yields a good performance for all workloads. But in general, an internal coding block size of 4 kB is a good compromise for performance reasons, and the proposed scrambling scheme seems to be a considerable

alternative to strong codes. At this point a theoretical analysis of the reliablity of weak codes combined with scrambling is required.

The work in this paper discussed the challenges of improving single disk reliability and gave some hints how to address them. In future work, we will extend our benchmarks with more energy efficient main boards and additional disks and evaluate how IDR affects the performance of different RAID schemes.

4 LoneStar RAID: Massive Array of Offline Disks for Archival Systems

Abstract

The need for huge storage archives rises with the ever growing creation of data. With today's big data and data analytics applications, some of these huge archives become active in the sense that all stored data can be accessed at any time. Running and evolving these archives is a constant tradeoff between performance, capacity, and price.

We present the LoneStar RAID, a disk based storage architecture, which focuses on high reliability, low energy consumption, and cheap reads. It is designed for MAID systems with up to hundreds of disk drives per server and is optimized for "write once, read sometimes" workloads. We use dedicated data and parity disks and export the data disks as individually accessible buckets. By intertwining disk groups into a two-dimensional RAID and improving the single disk reliability with intra-disk redundancy, the system achieves an elastic fault tolerance that can at least recover all 3-disk failures. Furthermore, we integrate a cache to off-load parity updates and a journal to track the RAID's state. The LoneStar RAID scheme provides a MTTDL that competes with today's erasure codes and is optimized to require only a minimal set of running disk drives.

4.1 Introduction

The need for huge storage archives rises with the ever growing creation of data. While tape is still widely used for archiving, it has its disadvantages and is not suitable in all scenarios. As it comes with high latencies and narrowed parallel access limited by the number of drives, purely tape-based solutions can, for instance, not provide the access times required by many active archive applications like the *Internet Archive* [JK09] or the archive of the ECMWF [GNM+15]. In such near-online systems, every set of data has a low probably to be requested, but still needs to be accessible within a short time frame. For this reason, the huge tape-based active archive of the ECMWF is enhanced by an efficient disk-based cache which serves almost all requests. However, as long as tape is included, the *time to first byte* can always add up to multiple minutes. In this work, we propose a storage system consisting of hard disk drives only. It is tailored for "write once, read sometimes" scenarios as they can be found in the aforementioned systems.

Leventhal argues that classical RAID 6 systems with double parity are no longer resilient enough for today's storage systems [Lev09], as probabilities of latent sector errors and rebuild times grow with the number and capacity of disk drives in a storage system. Hence, upcoming large-scale storage systems have to integrate stronger reliability mechanisms to cope with data loss. The situation can be improved by using declustered RAIDs [HG92] because they have faster rebuild mechanisms. As an alternative, multiple replicas of a file can be stored, which results in additional storage capacity that has to be bought and powered. Modern cloud-based storage systems store a single copy of a file and use erasure coding to be able to compensate the loss of multiple disks with a low storage overhead of 1.33x [HSX+12a, SAP+13]. Data is erasure-encoded and spread to $n + k$ blocks, where any n of them are required to decode the data object. To achieve the best reliability, these blocks have to be spread to $n + k$ different error domains, like independent disks.

For workloads where data is rarely accessed, it can be worthwhile to spin down disks to save energy [CGN02], but the aforementioned erasure-coded storage as well as declustered RAIDs assume that all disks are available and running at any point in time. All IO operations involve multiple disks as data can be striped or multiple chunks of data have to be read to calculate data. Hence, spin-down mechanisms are not viable.

The most energy can be saved by using simple computations that create resilient codes and storage mechanisms that reduce the number of required spinning disks. The presented work is an improved architecture and a description and analysis of the LoneStar RAID [GMPB11]. This system is optimized for "write once, read sometimes" workloads and aims for a high reliability while minimizing the overall energy consumption. The proposed system can be used for both, cold as well as active archive environments. It is optimized for cheap and fast reads as only a single target disk is required to be spun up. Writes on the other hand require multiple disks to update related parities. We present a new journaling approach and a parity cache that allows the system to defer and merge parity updates while keeping a high reliability and recoverability during updates.

Each disk uses intra-disk redundancy and is backed by multiple parity groups that intertwine into a two-dimensional RAID array. This setup can recover from many media errors, like corrupt blocks or irrecoverable read errors, and provides an elastic fault tolerance that is at least 3-disk fault tolerant (3-DFT). This means that it can at least recover from every combination of 3 simultaneous disk failures. Furthermore, the intertwined RAID scheme can also compensate many combinations of more than 3-disk failures.

The remainder of this article is organized as follows. After discussing the related work, we describe the general idea of the LoneStar RAID scheme in Section 4.3 including the overall architecture and key concepts like caching and journaling. In Section 4.4, LoneStar will be evaluated regarding reliability and performance. We investigate the *mean time to data loss* (MTTDL) and analyze different RAID configurations for multiple workloads. The work is summarized and concluded in Section 4.5.

4.2 Related Work

Many different aspects of data archival have been explored that span architectures for global distributed storage systems, designs of single storage servers, and optimizations for accesses of single disk drives.

Especially for archival systems, the reliability and the energy-consumption of a single storage server are crucial. In general, disks are spun-down as long as possible as argued by Colarelli et al. [CGN02], and to provide a high reliability, various codes are used to build redundancies over multiple storage nodes or disks.

The Pergamum system by Storer et al. [SGMV08], for example, uses strong erasure codes and spreads data over multiple storage nodes. By aggressively turning off these nodes, the system yields a promising energy efficiency while providing a high reliability.

Other systems build on multiple disks in one system. Schwarz and Burkhard, for example, investigated multi-dimensional disk arrays [SB93]. They analyzed the MTTDL of the whole system and also considered the power supply and SCSI controllers next to typical media errors.

The concept of two-dimensional RAID arrays in archival systems was also reinvestigated by Pâris et al. who use an additional XORed super parity disk, once all disks of a parity group have been written [PSAL12].

The Panasas Tiered Parity architecture is an example for a system that integrates multiple redundancy schemes to achieve a high reliability [JRU+10]. A RAID 5 spans multiple disks that use intra-disk redundancy and additional parities that span multiple storage nodes make the system robust against node failures. This system targets reliable systems that require a high performance.

Stodolsky et al. investigated the idea of delegating parity updates to an intermediate storage. To overcome the performance penalty of small writes in a RAID system, they hold parity updates in a fault-tolerant buffer. When enough data is gathered, these buffers are flushed with a large sequential write to the parity disk [SHCG94].

Thomasian and Blaum provide a broad overview of different RAID schemes [TB09]. Some energy conservation techniques for disk array systems have been proposed by Pinheiro and Bianchini [PB04]. For example, they presented the popular data concentration (PDC) approach that aggregates often used data to a small subset of the disks while spinning down the remaining array.

Felter et al. proposed a reliability-aware hybrid storage system that combines power-aware flash caching and disk spindown [FHC11]. To limit the rate of disks beeing powered on and off, they use a token system that limits the power cycles and reduces hardware wearout. To reduce the IOs on the disks, they introduce a SSD as a cache for the disks.

Narayanan et al. proposed a mechanism to off-load writes to a smaller set of disks to be able to spin down larger sets of disks [NDR08].

To reduce the energy consumption of RAID based systems, Xiao et al. presented Semi-Raid, which uses modified RAID 5 strategies and is optimized for sequential workloads [XYAZ11]. It sacrifices part of the parallelism of RAID in order to save power by activating only a few disks in the array and putting other disks into standby mode while still providing enough transfer rate for the application. Li and Wang presented EERaid, which also targets RAID 1 and RAID 5 schemes [LW04].

Wang et al. introduced eRaid, which uses caching and redirecting of data to spin down mirror-groups of disks to reduce the energy consumption [WZL08].

Weddle et al. presented PARAID, which shifts between multiple RAID levels (gears) to match different performance requirements [WOQ+07]. For a low gear, data is redirected to a small set of disks providing a RAID scheme. The other disks may be spun down. For higher gears, more disks are spun up and the RAID scheme is distributed over all active disks.

Greenan et al. tried to reduce the overall energy consumption by using dynamically configurable erasure codes that depend on the number of active disks [GLM+08]. The encoded blocks are written to the set of currently running disks, and in case of a read access, they assume that enough disks are currently spinning to reconstruct the data from the online disks. In later work, Greenan et al. explored various MDS codes and analyzed their efficiency and rebuild costs [GLW10].

To achieve performance and resilience, storage architectures like HDFS, Ceph, or OpenStack Swift store multiple replicas of objects to independent disks and locations. Despite beeing simple and robust, the huge storage overhead is a serious factor in terms of server hardware and energy consumption as each additional copy needs to be stored and maintained. To reduce this overhead todays object storage systems move towards erasure coding, which still is subject of performance improvements [PGM13]. To reduce the rebuild overhead of erasure coded storage systems, Huang et al. proposed Local Reconstruction Codes (LRC) that reduce the number of required coded fragments to recover data [HSX+12a]. These mechanisms were also applied to big data platforms to reduce the amount of data [SAP+13].

Balakrishnan et al. proposed Pelican, a right-provisioned disk-drive based cold storage system [BBD+14]. A Pelican building block consists of two servers that control 1,152 disk drives and schedule accesses to them. The system is provisioned to saturate the server's network links, thus saving superfluous hardware and reducing the overall energy consumption. Disks are managed in failure, power, and cooling domains and accesses to the disks are scheduled into waiting queues, as only up to 96 drives can be powered at the same time.

Thanks to huge studies of large disk drive populations [BGPS07, BADAD+08, SG07] and their analytical evaluation by Schroeder et al. [SDG10], we now have a good understanding of error properties of single disks and know how to cope with typical disk drive errors. The impact of irrecoverable disk errors on disk array reliability was first explored by Dholakia et al. and further investigated by Pâris et al. [DEH+08, PSLA08, PALS09].

4.3 LoneStar RAID

The goal of the LoneStar RAID architecture is to save energy by aggressively powering down unused components and reducing the number of required disks to a minimum, while providing reliable storage with a focus on read performance. Given a storage server with 192 disk drives and the disk power consumption measurements from [GMPB11] of 5.23 W for an idle and 2.19 W for a sleeping disk, the base consumption of an idling and immediately responsible disk array is 1004.16 W while the sleeping disk array, which requires to spin up disks on access, requires 420.48 W. The LoneStar MAID is designed to physically power off disks which reduces the disk energy consumption during idle times to 0 W. Spinning up a disk takes time, wears out its mechanics and already uses more than 20 W before it is accessible. Therefore, we aim to minimize the number of required disks for read and write operations and keep them spun down as long as possible.

To achieve a high reliability, the disks are arranged in a two-dimensional RAID where each disk is part of two RAID 4 schemes. Furthermore, by using intra-disk redundancy on each disk, corrupt blocks and irrecoverable read errors can be detected and corrected during accesses.

We do not stripe data to multiple disks and export one huge logical disk but use dedicated *data* and *parity* disks and export all data disks as individually accessible *buckets*. Hence, a read request requires a single spinning disk. Writes, on the other hand, require updates to 4 disks and a full read-modify-write (RMW) cycle on each of them. To recover data, a full parity group has to be started and read.

The LoneStar system provides a block-level API that can be embedded into higher-level applications or interfaces. To ensure a consistent state we use a synchronously written journal to track ongoing operations. By introducing a fast non-volatile cache into the system, updates to parity disks can be delayed and marked in the journal. When a disk spins up, all outstanding writes are merged and persisted, and when all relevant disks are

updated, the promise is marked finished. Until the data is flushed to the disks, the caches also work as a read cache.

4.3.1 Intra-Disk Redundancy

Previous work by [BGPS07, BADAD+08] and an analysis of [SDG10] show that latent sector errors are a serious threat for highly reliable storage systems. One mechanism to cope with these errors is intra-disk redundancy (IDR), where for n data blocks k parity blocks are calculated and stored together as a parity group. These additional k blocks then provide mechanisms to verify read data and, in case of a sector error, to recover up to k bad blocks in the parity group. Many different codes that could be used for IDR have been proposed [DEH+08, WT07, Pla08b, Haf05, SDG10] and we evaluated a selection of them to analyze the impact of applied intra-disk redundancy on both disk and CPU performance and the resulting energy consumption of the full storage system [GSB+11].

Schroeder et al. present a new failure model for disk drives and carve out the burstiness and correlation of errors [SDG10]. They also show that even a simple XOR based code like single parity check (SPC) strongly improves the reliability of a single disk.

We build on these results and add a *scrambling* layer to improve the reliability of simple codes [GSB+11]. Here, a set of s contiguous parity groups ($n+k$ blocks each) is intermixed to maximize the physical distance between contiguous blocks of the parity groups. With a $n = 16, k = 1$ code, a scrambling of $s = 10$ enables the code to compensate an error burst of up to 10 physically contiguous blocks for a small overhead for sequential workloads.

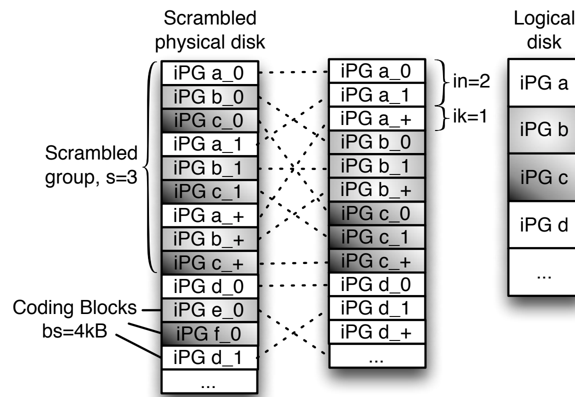


Figure 4.1: Scrambled on-disk layout and resulting iPGs for $s = 3, in = 2, ik = 1, bs = 4\,\text{kB}$.

We define the smallest logical unit that can be used on the disk level as an *intra-disk parity group* (iPG) that consists of $in + ik$ coding blocks. As evaluated in previous work, we propose to use a block size $bs = 4\,\text{kB}$ [GSB+11]. We tested multiple encoding schemes with block sizes of 512 Byte, 4 kB, and 64 kB and found 4 kB to be the best performing block size.

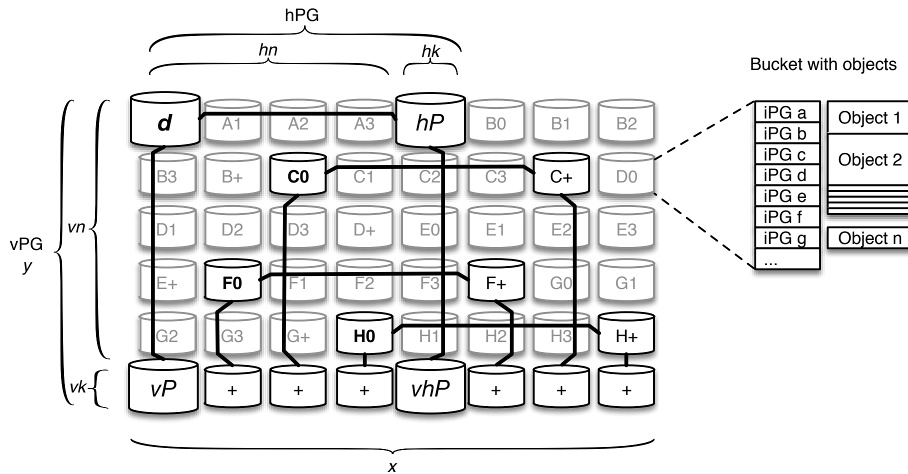


Figure 4.2: LoneStar RAID for d=48 disks with x=8 y=6, dn=32, dk=16, hn=4, hk=1.

By choosing a simple and computational inexpensive IDR code like the single parity check (SPC), the compute and storage overhead and therefore the resulting energy consumption is very low. Furthermore, even a slight improvement of the single disk reliability will improve the overall reliability of the system and reduce the required frequency of other mechanisms like disk scrubbing [IHHE08]. In the following, we use a 16+1 single parity check (SPC) and 4 kB block size. Therefore, the iPGs have a size of 68 kB containing 64 kB of data (see Figure 4.1).

The LoneStar RAID can integrate stronger IDR codes, but we found the SPC to be a good compromise between reliability, compute and storage and overhead. Especially, as the IDR is integrated into a strong RAID environment.

4.3.2 Intertwined RAID Scheme

Figure 4.2 shows a schematic overview of the two-dimensional RAID that can be fully described with the parameters x, y, hn, hk, and the used IDR code. The array consists of $d = x \cdot y$ disks with $dn = 32$ data disks and $dk = 16$ parity disks. The columns of the array build the *vertical parity groups* (vPG) with parameters vn and vk. These RAID 4 schemes consist of $vn = y - 1$ data disks and $vk = 1$ parity disks, where the parity disks are aligned to the last row. Analog to that, the parameters hn and hk describe the *horizontal parity groups* (hPG) with hn data and $hk = 1$ parity disks. We require $hn + hk < x$ so that no two disks of an hPG can belong to the same vPG. Each disk in this RAID is enhanced by IDR and their access granularity is a full iPG.

A read request can be as simple as spinning up the target bucket, reading and returning the requested data portion of one or multiple iPGs. In addition to that, the intra-disk redundancy can be used as a checksum scheme by recalculating and comparing the iPGs' parity blocks (Check on Read). If this fails, or to provide even more confidence, the bucket's horizontal and/or vertical parity groups can be started to recover and check the requested iPG.

Writes, on the other hand, have to pass three phases as depicted in Figure 4.3. In the *read phase*, a group of consecutive iPGs (iPGGroup) is read from all involved disks and optionally checked for its integrity using the iPG's parities. During the *coding phase*, the new iPGGroup's parity is calculated. The result is XORed with the bucket's old data into the *delta*-iPGGroup. This delta can optionally be cached and is used to update the parity disks. The *write phase* then writes the data to the disks.

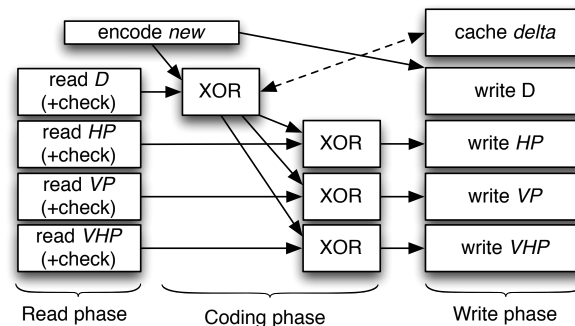


Figure 4.3: Pipeline to write *new* data to the RAID. The cache step is optional.

The process can be parallelized to multiple threads, but some dependencies within the process flow have to be met. The order of execution and the use of a journal and a cache to support the process heavily decides on the general throughput, the reliability of the whole system as well as the resulting energy consumption. Especially, since all disks are expected to be spun-down or may be unavailable for some time should not stop the system to accept newly written data.

In contrast to previously published approaches for two-dimensional RAID systems that use a list of aligned horizontal RAID 4 schemes with an additional vertical RAID 4 [SB93], the proposed shifted scheme adds an improved recoverability. For the classic scheme many error patterns can be found where the simultaneous error of 3 disks leads to a data loss. By shifting the horizontal parity groups, all combinations of simultaneous 3 disk failures can be tolerated and recovered.

By adjusting the LoneStar scheme's parameters, the storage efficiency and reliability as well as the achievable parallelism and therefore performance can be fine tuned. By choosing small vertical or horizontal parity group sizes, the rebuild of a single failed disk would become cheap as only one parity group has to be spun up for the rebuild process. By choosing huge vPGs or hPGs, the achievable parallelism and therefore the overall throughput will shrink as locks span both the full vPG and hPG of a bucket. Furthermore, the typical parity disk write bottleneck of RAID 4 based systems reduces the throughput. If both, the hPG and vPG are big, a high storage efficiency is achievable but the reliability, the rebuild overhead, and the RAID 4 based write congestion have to be considered. By exploring the parameter space of possible configurations, many different configurations can be found that favor the aforementioned criteria. Tables 4.3 and 4.4 will present filtered lists of possible configurations and Section 4.4.3 discusses their characteristics.

The parameterization of the IDR scheme also affects multiple metrics. An IDR configuration $in + ik$ reduces the overall storage capacity of the system by an additional factor of $\frac{in}{in+ik}$ but improves the overall robustness against media errors and reduce the number of required RAID rebuilds to recover from bad disk sectors. A system with dn data disks and dk parity disks has a factor of

$$r = \frac{in}{in + ik} \cdot \frac{dn}{dn + dk}$$

for the available capacity. For our prototype evaluation and the following discussion we use a block size of 4 kB and a $in = 16, ik = 1$ SPC code resulting in iPGs of 68 kB size.

4.3.3 Journaling & Promises

Given the intertwined nature of the LoneStar RAID, any read or write access to any bucket can interfere with parallel operations to dependent disks. While multiple overlapping read requests can be executed in parallel, writes need to handled with care. Writing a single byte to a bucket moves the system to an inconsistent state that needs to be restored again. This can be done by either propagating the changes to the depending parity disks, or by restoring the disk's previous state. In order to attach a traceable state to each of these changes, we organize RAID accesses into write promises and track their state in a journal. We define a write promise to be a state of an ongoing write operation that exactly defines the affected range of iPGs on a bucket and the state of the 4 write operations to the target and the related parity disks. The promise's unique *txn_id* is used as the primary key for journaling and caching operations and provide a reference point for accessing applications. In contrast to transactions known from databases, we do not provide a rollback that recovers the previous data but guarantee that aborted or failed promises are further processed until the RAID reaches a consistent state again.

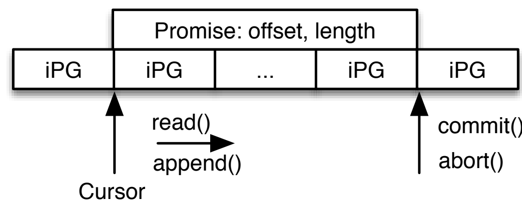


Figure 4.4: Buckets are accessed via forward-only read- and write promises.

To access data on the LoneStar RAID, an application creates a read or write promise for the target bucket and range. Then, it can read bytes from or append new data to the promise object. This forward-only approach as depicted in Figure 4.4 not only provides a simple API but also builds the foundation for a memory efficient, asynchronous, and highly parallel core to process the I/O requests.

Internally, the LoneStar RAID uses a group of consecutive iPGs (iPGGroup). Such a group can contain a single iPG of 68 kB, or aggregate multiple logically consecutive iPGs. These groups are read from or written to the RAID in compound operations. When a

read promise is created, a request to load a first iPGGroup is scheduled, but many more requests can be started for a read-ahead prefetching. For write promises, appended data is aggregated into an iPGGroup that is then processed at once. The RAID then acquires a lock for this group, and processes it. To fulfill all requests of a promise, multiple read and write requests may be started. These processes each consist of multiple asynchronous and parallel operations and include the reading, encoding, checking, writing and transparent recovery of data.

In general, write promises can be aborted. If not all parity disks have been updated, the previous data could be restored. But as this is an unclear behavior, the already scheduled write tasks have to be finished to get the system into a consistent state. To abort, the current promise's append buffer is discarded, the promise is closed and a consistent system state is restored.

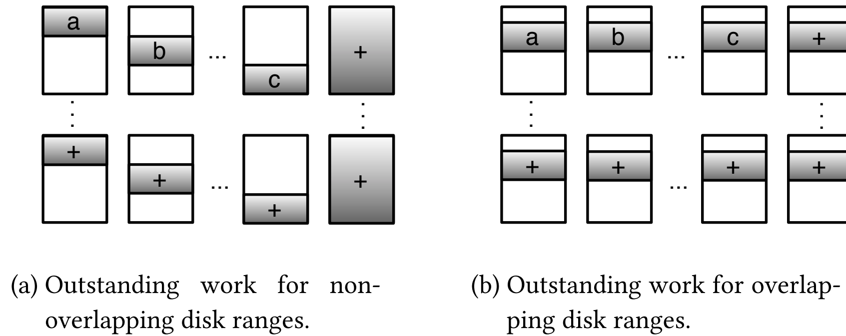
To overcome the challenges for consistency and reliability we introduce a journal to keep track of ongoing write promises and outstanding operations. The journal is synchronously written to a fast, reliable non-volatile RAM. Similar to previous systems like WAFL [HLM94] that write a replayable journal of both, operations and contents, we use the journal to persist a list of outstanding operations while intermediate data is written to a dedicated cache. For every write promise a new journal entry is created that contains the promise's unique txn_id, target disk id, offset and length of the operation, and the disk ids of the corresponding target parity disks hP, vP, vhP. For each of the disk ids two additional time stamps are stored that mark the start and finish times of operations. These time stamps can be seen as a list of outstanding operations. When the write promise is created, all time stamps are initialized with 0. Whenever any disk starts to work off the promise's data, the respective timestamp is set. When all disks have marked the promise as finished, the promise's data is deleted from the cache and it can be removed from the journal.

4.3.4 Caching

The intertwined RAID schemes alongside the properties of the used RAID 4 schemes render two bottlenecks during parallel accesses to the system. As every operation covers ranges of iPGs over multiple independent disks, a write operation may have to wait for other operations to succeed. A second bottleneck is introduced by the RAID 4 parities. Even if writes to different disks of one horizontal parity group do not lock each other, the parity disk hP defines the performance of the full horizontal group. During parallel access, this RAID 4 bottleneck might even be amplified because of the shared vertical parity disks as depicted in Figure 4.2.

To alleviate the impact of the aforementioned congestions due to parallel accesses and to overcome the need that all target disks have to be spun up for writes, an additional caching layer is introduced. The dependency graph shown in Figure 4.3 presents the delta that can be cached. The previous iPGs from the data disk are XORed with the new data and the resulting data is either processed by the parity disks or stored to the cache. By

not requiring all dependent disks for updates, the overall throughput can be improved and multiple writes to parity disks can be merged to reduce the amount of written data and therefore to reduce the required energy.



(a) Outstanding work for non-overlapping disk ranges.

(b) Outstanding work for overlapping disk ranges.

Figure 4.5: By aggregating writes to overlapping ranges of data disks, the actual work on the parity disks can be reduced.

Figure 4.5 shows two scenarios where multiple disks of a parity group are written. Figure 4.5a shows updates to non-overlapping ranges of the disk. While caching can help to aggregate writes to the parity disks, the horizontal parity disk (hP) and its vertical parity disk (vhP) each have to process the same number of iPGs as the data disks combined. Like direct writes, to update a single iPG on a data disk, in total 4 iPGs have to be read and 4 iPGs have to be written. This overhead factor of 8 can be reduced as shown in Figure 4.5b. Here, the same range on all data disks of a horizontal parity is written. The deltas are cached and can be merged before being written to the parity disks.

Given a parity group with hn data- and hk parity disks, where the same range on all data disks is written and all updates are cached and merged, then all dependent hn data disks, hk horizontal parity disks (hP), hn vertical parity disks (vP) and hk vhP disks each execute a full read-modify-write cycle on the same number of iPGs. This results in a factor of $(2 \cdot (2 \cdot hn + 2 \cdot hk))/hn$, which is is 4.57 for a (7+1) parity group.

To further reduce this factor, a combination of data and parity disks have to be found where the ratio of data to parity disks is high. This is the case when multiple data disks share the same set of parity disks, but due to the alignment of the disks, no two data disks share the same set of vertical parity disks. As a result of this, the best ratio of data to parity disks is given, when all data disks are written in parallel. For d data and p parity disks the overhead factor can be reduced to $(2 \cdot (d + p))/d$, when the same range on all data disks of the array is written and all updates can be cached and merged. For example, a $x = 8, y = 8, hn = 6, hk = 1$ configuration has an optimal factor of 2.67.

We combine caching and journaling to delay writes to the parity disks. The cache is organized as a key-value map and is persisted to a reliable, non-volatile RAM. The size of the cache determines how many iPGs can be buffered and how long writes to parity disks

can be delayed. The key of a cached entry is the txn_id and the value contains the delta that needs to be applied to the parity disks.

If a promise is created for delayed writes, its txn_id is registered with the cache. If the cache is full, the creation can either wait for the cache or fall back to a direct write mode that does not cache any data. When the promise finishes, so that all data hit both the target data disk and the cache, the txn_id is marked to be available for further operations.

At some point in time, the cached promises have to be applied to the parity disks. This happens within a *Cache2Disk* task that operates on all outstanding txn_ids for a single parity disk. The task aggregates some or all promises, and marks their journaled state as "processing" by setting the appropriate start timestamps. Overwritten iPGs as well as write operations to data disks within a horizontal or vertical parity group may result in overlapping promises that need to be written to parity disk. In a first step, the *Cache2Disk* orders all promises and aggregates them into combined tasks. If overlapping data is found, it is merged before being XORed with the parity disk's old data and being persisted to the disk. When the write finishes, the promise's finish timestamps are updated in the journal.

The cached data always overlays the data of the parity disks. If, for example, a recovering process reads data from a parity disk, the cache is always queried. If it contains data for the requested iPG range, the cached promises that cover the range are first applied to the parity disk before the requested iPGs are returned.

There can be various policies to decide when to start a *Cache2Disk* task. Generally, if a parity disk is online and has cached promises, a *Cache2Disk* task is started. Disks are aggressively powered down and parity disks are only required for uncached direct writes, and recovery operations. If there are no errors or external maintenance tasks that start up disks in regular intervals, the parity disks rarely need to be started.

Here, a trade-off has to be achieved. On the one hand, the reliability of cached data is reduced, and a full cache can add some latency for newly written data, as other promises have to be flushed before new promises can be accepted. On the other hand, cached promises can be aggregated, merged and written to the parity disks at once. This can result in less data to be written, and a higher throughput, as all promises are ordered and can be persisted in one sequential write operation. Furthermore, disks need to be spun up less often, which results in a reduced energy consumption.

4.3.5 Write Modes & Semantics

When an application creates a write promise, it can specify some options that influence the overall performance and reliability of the update. Referring to Figure 4.3 and the previously presented journaling and caching mechanisms, multiple write modes are possible that differ in reliability, performance, and energy consumption.

Direct Write

For direct writes, the data disk and all three parity disks are started, when the promise is created. No data is cached and all disks are read and written as soon as possible. When the promise's metadata is written to the journal, all start timestamps are set. On return of

the promise's `commit()` method, the new data is guaranteed to be persisted to all disks and no further operations are required.

The major drawback of direct writes is the recoverability during an error. If an error occurs during the operation, all disks are in an undefined state and need to be recovered. This condition hits the RAID write hole problem and triggers the irrecoverable EP2 error pattern presented in Figure 4.6.

Serialized Direct Write

To overcome the RAID write hole of the direct writes, the updates to the parity disks can be serialized. The update processes to the 4 target disks are staggered so that a particular range is only updated on a single disk and therefore do not interfere with each other. First, the data disk is written, and on completion the data disk's finish time and the parity disks starting time are updated. Then the parity disks are updated one after the other with updates to the journal in between. By executing the writes in a strict order, on error only a single disk is in an undefined state which results in a defined recovery.

In contrast to the direct writes, the serialized operations result in a higher execution time until the promise is committed. On the other hand, multiple promises can be active in parallel on the disks, which theoretically results in the same overall throughput of the system. When `commit()` is called, the system can either return, when the data disk is written, or all parity disks finished their writes.

Delayed Write

The caching and journaling mechanisms provide means to always have a consistent view of the RAID's state and allows the system to delay writes for long times until the promise's data are fully persisted.

When a promise is created with the delayed mode, a `commit()` will return as soon as the data was written to the target disk and the delta hits the cache. As the journal tracks every step of the promise, callbacks can be provided to notify subscribers on ongoing state changes. A use case for these callbacks is an application that writes some data to the LoneStar RAID. When all files are fully persisted, and therefore provide the highest available fault-tolerance and reliability guarantees, the application is notified.

4.3.6 Recovery & Maintenance

The LoneStar RAID integrates multiple mechanisms to maximize the reliability of the data. This includes mechanisms to detect and compensate errors on single disks and the compensation of simultaneous failures on multiple related disks. Furthermore, the journal provides an exact view on failed operations and can track the progress of recovery operations.

In contrast to simple common RAID schemes that need to rebuild and recover the full RAID in case of an error, the LoneStar RAID always exactly knows the logical state and the history of the full RAID array. On system start, the journal is loaded and in case of an unclean shutdown, the last known state of the disk is known. If any promise was started on the disk but does not contain a finished entry, the disks range is marked as defect

and is recovered from the RAID. The journaling also implicitly eliminates the write hole problem where data is written to disk but the parity updates failed. This condition can be detected in the journal and recovered by pushing outstanding writes to the parity disks or by restoring the information on the data disk. In case of an irrecoverable error the failure is isolated as the exact range of erroneous blocks on the data and parity disks is known. Then, a neutral, consistent state can be created for that range.

Whenever a full disk or parts of a disk are marked as defect, the affected ranges are stored in the journal. If a disk is replaced, the full disk's range is marked as defect. When the disk is recovered, a single background process sequentially recovers the defect ranges. The next available defect range of up to 64 MB is acquired, locked, and in case of successful recovery, removed from the defect ranges in the journal. If another process tries to access data that is not yet recovered, the range is locked, recovered in place and the journal is updated. Knowing the exact state of the disk's health can also be used to delay recovery operations.

If an iPG was successfully read from a disk, it can be checked on read by recalculating its ik parity blocks. This calculation can be used as a checksum that can detect corrupt blocks. If the calculated value equals the parity blocks, the iPG is returned. Depending on the used IDR code, the corrupt block may be determined, marked as defect, and corrected by the IDR scheme. If an I/O error occurs during a read, the remaining blocks of the iPG are tried to be read and the defect block is recovered by the IDR scheme. If this fails, or the recovered block should be rechecked for integrity, the data is recovered from a horizontal or vertical parity group.

When an iPG is marked as defect, or a full disk fails, the surrounding RAID is used for recovery. For the rebuild the smaller intact parity group (VPG/HPG) needs to be spun up, and the rebuild could run at the full disk speed as all but the failed disk read only and the recovering disk writes only. A write lock is acquired on the affected iPGs and for all reads the same mechanisms as for reading an object apply.

During a rebuild process multiple additional disks may fail or have irrecoverable read errors. If independent disks fail, individual recovery processes are started. When data is recovered, the data of the parity group's disk is read, encoded and written back to the disk. When another disk of the parity group faces irrecoverable read errors during this process, the read request transparently blocks until the disk is recovered from its other parity group. If another disk fails within this recovering parity group, a third recovery process will be started. If multiple disks in dependent recovery groups fail, data may become unavailable but it still can be recovered. Although the intra-disk redundancy and the surrounding RAID provide a high reliability, undetected media errors and failed disks can at least lead to additional recovery overhead and temporal unavailability of data.

By offloading parity updates to a cache and persisting the temporal logical state to an additional journal, the system's reliability during update operations has to be reconsidered. A hard error during a direct write puts the system in an irrecoverable state as 4 interdependent disks are written in parallel. In contrast to the error patterns presented in

Figure 4.6, this condition can be recovered. As the promise did not return from the commit, the system does not guarantee the data to be persisted. In this case, the journal is queried for the unfinished promise's range and all three parity disks are marked to be recovered.

To recover from errors that occur during a serialized direct write, the same mechanisms as for direct writes can be applied. Nevertheless, only the state of one disk is left in an undefined state. If the write to the data disk was not yet finished, the promise was not yet committed and the data disk needs to be recovered. If, on the other hand, the data disk was successfully written, its updated data needs to be pushed to the parity disks. As for the direct writes, if not yet updated, the parity disks are marked as erroneous and a recovery process is started.

If the journal is damaged, no data is immediately lost, but a very expensive recovery process is required to guarantee a consistent state of the RAID. All parity disks are marked as possibly erroneous and their contents need to be recovered and checked. As all committed writes have hit their target data disks, outstanding updates will be pushed to the parity disks in this way.

Also, the loss of the cache will not result in a data loss, as all cached data also already have hit the data disks. The journal can look up the outstanding writes. The outstanding parity disks' ranges will be marked for recovery and the data gets fully persisted.

To overcome the problems of aging media and silent data corruption disk scrubbing was subject of active research [SXM+04, IHHE08, PSAL10]. Nevertheless for a system where most of the disks are powered off most of the time, a naive disk-scrubbing approach results in a waste of energy as disks are spun up only for being checked. A feasible solution to this problem was found in Pergamum [SGMV08], where parts of disks are checked before they are put to sleep. This concept can be integrated very well in the LoneStar RAID approach. Whenever a disk is started, some of its parts are checked. Here again, the journal helps to keep track of outstanding operations. If all disks are regularly used, over time all disks are fully scrubbed.

4.3.7 Performance

Generally, the performance of storage systems is measured by throughput for read and write operations and the related energy consumption. Due to the complex and intertwined nature of the LoneStar system, and the different system behavior based on the configuration's parameters it is not trivial to argue about these metrics. Many factors like the underlying hardware, the used configuration, the order of operations based on the write mode and the application's access patterns have to be considered.

First, the access latency has to be discussed. If all disks are powered off, every read or write request first has to wait until the disk drive is spun-up and ready for operation. Once, a disk is powered, the decision when to shut it of again influences the overall power consumption. [AMS10] underline this point by simulating the long-term behavior of disk based archival systems. Here, the disk's idle time before being shut down present a strong factor. [WM10] propose 50 seconds of idle time as a good trade-off for archival workloads.

The argumentation for read performance is straight-forward. All data disks are exported as individual buckets, and if the disk is not currently written by another process or part of a recovery, it can be accessed with nearly full hardware speed. Only the storage and compute overhead induced by intra-disk-redundancy and check on read has to be considered.

To recover bad iPGs or failing disks, the surrounding horizontal or vertical parity group has to be started. The recovery process itself reads from parity groups disks and XORs their contents. In case of a repair the recovered content is written back to the recovered disk. As no parity has to be updated and the recovered disks data is overwritten, both, the reads from the recovering disks and the writes to the recovered disk can operate at full hardware speed with an additional latency for the journal updates and the XOR computation.

To argue about the performance of write processes, some additional factors have to be considered. First, to maintain the logical consistency of the RAID, all newly written data have to be XORed with the previously written data. This means that for every write, a preceding read on the same disk is required. This read-modify-write cycle reduces the possible write throughput by a factor of $\frac{1}{2}$ compared to reads. Furthermore, if many operations are executed in parallel, locking and shared accesses to disks can slow down the overall performance.

If the direct write mode is used, all parity disks are immediately written. If, for example, all data disks of a parity group are written in parallel, the overall throughput will be defined by the parity group's parity disk. For direct writes, the best throughput is achievable, when all vertical parity disks are busy and one bucket is written in every horizontal parity group. Figure 4.2 shows such a situation where the four disks D, $C0$, $F0$, and $H0$ are written in parallel and the maximum performance of the disk subsystem can be achieved. In general, the maximum throughput of direct writes to the RAID is given by

$$\min(\lfloor \frac{x}{1+hk} \rfloor, \text{\#HPGs}) \cdot \frac{\text{disk_write_throughput}}{2}$$

with a total number of horizontal disks x, hk parity disks in a horizontal parity group, and a $\frac{1}{2}$ factor for the RMW cycles.

For delayed writes on the other hand, the I/O bottleneck is moved from the parity disks to the cache. The cache has a much higher throughput and lower energy consumption than disk drives and does not need to be spun up before operation. The cache itself can be useful to intercept bursts, for example, when multiple disks of a parity group are written in parallel. Furthermore, if the same ranges on related disks are written, the cached updates can be merged, which results in less data that needs to be written.

As the cache's size is limited, it can only support continuous writes for some time. If, for example, a multiple of the cache's size is written to the buckets, the overall throughput will be constant until the cache starts some *Cache2Disk* tasks in the background or is forced to flush its contents.

If the throughput of a single bucket is not sufficient, an application can spread the data over multiple buckets. While the read performance will linearly rise with every additional bucket, the write performance and reliability of the chosen set of buckets needs to be

considered. Using all data disks of horizontal parity group creates the bottleneck of a shared horizontal parity group, but on the other hand uses a low number of involved parity disks. When data is spread to all buckets of a vertical parity group, also the shared vertical parity disk becomes a bottleneck, but all data disks also update individual horizontal parity groups which improves the recoverability of the data. The best performance but also the highest overhead is to write to data disks that do not share any parity disks.

4.4 Evaluation

In order to evaluate the proposed LoneStar RAID, we consider its reliability and performance and discuss its energy efficiency. The evaluation starts with the description of a reliability model that can be used to calculate the mean time to data loss (MTTDL) for storage systems with an elastic fault tolerance. To be able to compare results, we use a target scenario of an archive that can host 10 PB of user data. We analyze the reliability and behavior of LoneStar for multiple possible configurations as well as configurations of $n + k$ RAIDs, which could be realized with RAID 6 schemes or stronger erasure codes. Then, we describe a software prototype that is used to investigate various usage patterns of the system.

4.4.1 Reliability Model

We define multiple states the system can be in. If all components work correctly it is *functional*. It is *degraded* if one or more disks failed, but the system could compensate the failure of additional disks. If any additional error can lead to an irrecoverable state, it is in the *critical* state. Consequently, when at least a single bit is lost and could not be recovered by the RAID, the system is in the *data loss* state.

We are interested in the MTTDL of the system to be able to compare and argue about the system's reliability. For this reason, we will analyze the average time until the system reaches the *data loss* state. This only happens if a disk and disks of all available parity groups fail in a way that data could no longer be read or recovered. Based on collected failure data in a study on 1.53 million disks over a period of 32 months [BGPS07], Pâris et al. argue that the probability that the same regions on two disks in a parity group have an unrecoverable read error is insignificant [PALS09]. The fact that we further reduce the probability of a read error with the intra-disk redundancy allows us to simplify our failure model. This assumption is also backed by Iliadis et al. [IHHE08], who stated that a disk drive with intra-disk redundancy is almost as reliable as a disk drive without any sector errors. As argued by Gao et al., we skip all combinations of read errors on multiple disks and reduce the failure model to combinations of disks that fail completely [GMB10].

A transition to the state *data loss* can only happen if the system is in the *critical* state due to a series of irrecoverable block and disk failures. Then, if another block error, which cannot be recovered by the disk's IDR, occurs during the rebuild or if an additional disk fails, data may be lost.

We build on the work of Hafner and Rao who presented reliability models for non-mds erasure codes [HR06]. The LoneStar RAID scheme has the properties of such a code because it can tolerate many instances of k failures for $k = 1, \dots, t$, where t is the limit after which no more failures could be tolerated. This *upper threshold fault tolerance* is the number of parity disks dk and the *Hamming fault tolerance* is 3.

To model the "many combinations", we refer to the conditional probability p_k, which is defined as the probability that the erasure code can tolerate an additional disk failure, given that it has failed and tolerated k disk losses [HR06]. We are interested in the evaluation of huge disk arrays, where many disks may fail. To calculate p_k, we have to check for all $\binom{d}{k+1}$ possible combinations of simultaneous k disk failures whether all disks can be recovered by the RAID. In previous work, we used a brute force simulation approach to investigate for each combination if it could be recovered [GMPB11]. This approach was exact, but did not scale up to configurations with lots of disks d and $k > 5$. To calculate the

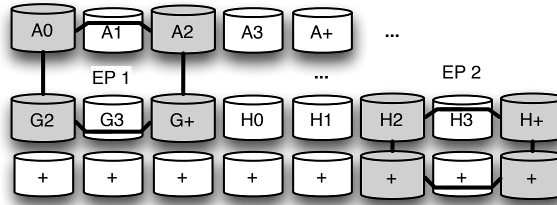


Figure 4.6: Examples for irrecoverable error patterns that lead to data loss. All other patterns of simultaneous 4-disk failures can be recovered.

probability of recovering from a k-disk failure, we analyzed the RAID scheme's conditions that lead to a data loss. Figure 4.6 depicts the two patterns that cannot be recovered. *EP1* occurs, when two hPGs have at least two defect disks each, and four defective disks share two vertical parity groups. The second pattern *EP2* is a special case of *EP1*, where two defective vertical parity disks and one defective horizontal parity group lead to an irrecoverable constellation. We count the number of occurrences of the two bad patterns for a given RAID configuration as b.

With d as the total number of disks in the system, we use $n_k = \binom{d}{k}$ as the number of possible k-disk failure instances. Then, s_k is the number of such k-disk failure instances the RAID scheme can tolerate. The conditional probability p_k is defined as [HR06]:

$$p_k = \frac{s_{k+1}}{\binom{d}{k+1}} / \frac{s_k}{\binom{d}{k}}$$

For $k = 3$, the conditional probability for the LoneStar scheme can be reduced to:

$$p_3 = 1 - \frac{b}{\binom{d}{4}}$$

And for k $k \geq 4$

$$p_k = (1 - \frac{b \cdot \binom{d-4}{(k+1)-4}}{\binom{d}{k+1}})/(1 - \frac{b \cdot \binom{d-4}{k-4}}{\binom{d}{k}})$$

To determine the MTTDL of the RAID schemes, we use a continuous-time Markov chain with an absorbing *data loss* state (DL), see Figure 4.7. The model parameters are listed in Table 4.1. We use similar model parameters as Hafner and Rao but shift the capacity to 4 TB to reflect current disk capacities. The effect of the intra-disk redundancy on the hard error rate is contained in the adapted value $HER(IDR)$, which is ten times smaller than the unimproved hard error rate. Considering the rate improvements suggested in the literature [PSLA08, PALS09, SDG10], this factor is a rather conservative estimate. We also use $\mu_t = \mu$ to model a sequential rebuild of failed disks, which is true if only dependent disks are failing. Again, this is a conservative estimate because independent failed disks could be rebuilt in parallel. Furthermore, we use a journal to track and manage rebuilds. We rely on the transition probability model given in [HR06]:

$$\sigma_k = (d-k)\lambda p_k\{1 - (1 - p_{k+1})(d - (k+1))h\}$$
$$\delta_k = (d-k)\lambda\{(1 - p_k) + p_k(1 - p_{k+1})(d - (k+1))h\}$$

Table 4.1: Model parameters for MTTDL analysis

Label	**Description**	**Value**
d	Number of disks in the storage system	Variable
λ	Drive failure rate, or $\frac{1}{MTTF_d}$	$\frac{1}{500000} hrs$
μ	Drive rebuild rate, or $\frac{1}{MTTR_d}$	$\frac{1}{12} hrs$
h	Probability of an uncorrectable error during rebuild per drive read	$C \cdot HER$
C	Drive capacity	4TB
HER	Hard error rate, in errors per number of bytes read	$8 \cdot 10^{-15}$
$HER(IDR)$	Hard error rate, in errors per number of bytes read with an applied SPC(16+1) intra-disk redundancy code	$8 \cdot 10^{-16}$

4.4.2 Mean Time To Dataloss (MTTDL) Analysis

To compare LoneStar with other RAID schemes, we use the same reliability model to evaluate $n + k$ RAID schemes. We calculate the MTTDL for a single array with n data and k parity disks and extrapolate the analysis to systems that can store 10 PB of user data. An $n + k$ array is able to tolerate up to k-disk failures. A storage system that consists of

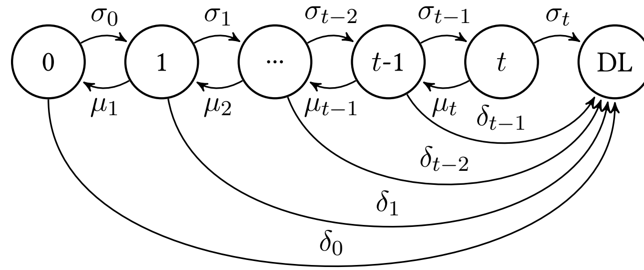


Figure 4.7: Reliability model for a storage array with non-MDS erasure code, based on Hafner and Rao.

multiple of these independent arrays is able to compensate many more simultaneous disk failures than k. Hafner and Rao found, that treating each array independently is a very good approximation of the MTTDL of the resulting full block of disks [HR06].

In the following, we calculate the MTTDL for an $n+k$ array. The conditional probability p_i is 1 for $i < k$ and 0 for $i \geq k$. The MTTDL for r independent arrays then is:

$$MTTDL_{r\cdot(n+k)} = \frac{1}{\frac{r}{MTTDL_{(n+k)}}}$$

Tables 4.2 and 4.3 present the results for various RAID schemes to host 10 PB of user data. The results assume 4 TB disks without IDR and with a fixed $16+1$ IDR scheme and present the resulting MTTDL, total number of disks, and the storage efficiency (Eff.).

The results show that the integration of intra-disk redundancy into the RAID improves the MTTDL of nearly all configurations by a factor of 10. While generally the storage efficiency is slightly worse, the MTTDL of the LoneStar RAID outperforms an array of $n+2$ RAIDs by multiple orders of magnitudes, and provides a similar storage efficiency and reliability as $n+3$ erasure codes. The reliability can be raised by further growing the parity groups to $n+4$ codes. They provide the highest MTTDL with a moderate storage efficiency, but also require the highest number of active disks per parity group.

4.4.3 Suitable Configurations

To compare some configurations for a high-density LoneStar-based system, we targeted systems with 192 disks. Today's RAID controllers, are capable of serving this amount of disks and modular disk-shelve building blocks can be used to build these systems. For example, a 12-disk server with three 60-disk 3.5" shelves attached, or a single 3U shelve with 192 2.5" disks can be used.

Table 4.4 presents ten suitable configurations with their storage efficiency, MTTDL, the number of horizontal parity groups, the number of non-overlapping parallel writes, and the number of disks required to restore a disk from their horizontal and vertical parity groups. In general, storage efficiency is traded for parallelism and reliability. Smaller horizontal and vertical parity groups result in a reduced efficiency but improve the overall

Table 4.2: MTTDL in hours for 10 PB user data built with multiple arrays of $n + k$ RAID schemes.

	with (16+1) IDR		**without IDR**	
$(\mathbf{n}+\mathbf{k})$	**MTTDL in hrs**	**#Disks (Eff. in %)**	**MTTDL in hrs**	**#Disks (Eff. in %)**
$6+2$	$3.86 \cdot 10^7$	$3,632$ (70.6)	$4.13 \cdot 10^6$	$3,416$ (75.0)
$7+2$	$3.00 \cdot 10^7$	$3,501$ (73.2)	$3.21 \cdot 10^6$	$3,294$ (77.8)
$6+3$	$1.79 \cdot 10^{11}$	$4,086$ (62.7)	$1.91 \cdot 10^{10}$	$3,843$ (66.7)
$8+2$	$2.40 \cdot 10^7$	$3,400$ (75.3)	$2.57 \cdot 10^6$	$3,200$ (80.0)
$7+3$	$1.25 \cdot 10^{11}$	$3,890$ (65.9)	$1.34 \cdot 10^{10}$	$3,660$ (70.0)
$6+4$	$9.59 \cdot 10^{13}$	$4,540$ (56.5)	$4.74 \cdot 10^{13}$	$4,270$ (60.0)
$8+3$	$9.09 \cdot 10^{10}$	$3,751$ (68.4)	$9.73 \cdot 10^9$	$3,520$ (72.7)
$9+2$	$1.96 \cdot 10^7$	$3,333$ (77.0)	$2.10 \cdot 10^6$	$3,135$ (81.8)
$9+3$	$6.81 \cdot 10^{10}$	$3,636$ (70.6)	$7.29 \cdot 10^9$	$3,420$ (75.0)
$10+2$	$1.64 \cdot 10^7$	$3,264$ (78.4)	$1.75 \cdot 10^6$	$3,072$ (83.3)
$8+4$	$8.81 \cdot 10^{13}$	$4,092$ (62.7)	$2.67 \cdot 10^{13}$	$3,852$ (66.7)
$11+2$	$1.38 \cdot 10^7$	$3,224$ (79.6)	$1.48 \cdot 10^6$	$3,029$ (84.6)
$10+3$	$5.26 \cdot 10^{10}$	$3,536$ (72.4)	$5.62 \cdot 10^9$	$3,328$ (76.9)
$11+3$	$4.11 \cdot 10^{10}$	$3,472$ (73.9)	$4.41 \cdot 10^9$	$3,262$ (78.6)
$10+4$	$7.14 \cdot 10^{13}$	$3,808$ (67.2)	$1.49 \cdot 10^{13}$	$3,584$ (71.4)
$12+2$	$1.19 \cdot 10^7$	$3,178$ (80.7)	$1.27 \cdot 10^6$	$2,996$ (85.7)
$13+2$	$1.03 \cdot 10^7$	$3,150$ (81.6)	$1.10 \cdot 10^6$	$2,955$ (86.7)
$11+4$	$4.13 \cdot 10^{14}$	$3,720$ (69.0)	$1.32 \cdot 10^{13}$	$3,495$ (73.3)
$12+3$	$3.30 \cdot 10^{10}$	$3,405$ (75.3)	$3.52 \cdot 10^9$	$3,210$ (80.0)
$12+4$	$5.29 \cdot 10^{13}$	$3,632$ (70.6)	$8.62 \cdot 10^{12}$	$3,424$ (75.0)
$13+3$	$2.67 \cdot 10^{10}$	$3,360$ (76.5)	$2.87 \cdot 10^9$	$3,152$ (81.2)
$13+4$	$9.31 \cdot 10^{13}$	$3,570$ (72.0)	$7.24 \cdot 10^{12}$	$3,349$ (76.5)

performance of the system as more groups can be accessed in parallel and recovery processes only require small numbers of disks.

4.4.4 Prototype & Simulation Environment

The current LoneStar RAID core consists of a C++11 based user space library that provides access to a set of virtual disk drives. These virtual disks can be local files, physical media like disk drives, or simulated objects. The library provides block-level access to the individual buckets via the promise API presented in Section 4.3.3. For the journal, we use a *redis* database that synchronously appends updates to an append-only file on a SSD. The cache data is stored on a regular file system on the SSD.

Every second a *WatchDog* task is started that checks the disks and the cache to apply policies, which decide when to persist cached data. If a disk is idle for a specific time, it is

Table 4.3: MTTDL in hours for 10 PB user data built with multiple LoneStar arrays.

		with (16+1) IDR		without IDR	
d	$\mathbf{x \cdot y, (hn + hk)}$	**MTTDL in hrs**	**#Disks (Eff. in %)**	**MTTDL in hrs**	**#Disks (Eff. in %)**
50	$10 \cdot 5, (7 + 1)$	$4.24 \cdot 10^{10}$	3,900 (65.9)	$4.49 \cdot 10^{9}$	3,700 (70.0)
64	$8 \cdot 8, (6 + 1)$	$3.59 \cdot 10^{10}$	3,648 (70.6)	$3.81 \cdot 10^{9}$	3,456 (75.0)
72	$9 \cdot 8, (6 + 1)$	$3.56 \cdot 10^{10}$	3,672 (70.6)	$3.81 \cdot 10^{9}$	3,456 (75.0)
75	$15 \cdot 5, (11 + 1)$	$1.22 \cdot 10^{10}$	3,750 (69.0)	$1.31 \cdot 10^{9}$	3,525 (73.3)
77	$11 \cdot 7, (5 + 1)$	$7.10 \cdot 10^{10}$	3,850 (67.2)	$7.60 \cdot 10^{9}$	3,619 (71.4)
88	$11 \cdot 8, (6 + 1)$	$3.54 \cdot 10^{10}$	3,696 (70.6)	$3.83 \cdot 10^{9}$	3,432 (75.0)
90	$18 \cdot 5, (7 + 1)$	$2.13 \cdot 10^{10}$	3,960 (65.9)	$2.30 \cdot 10^{9}$	3,690 (70.0)
91	$13 \cdot 7, (5 + 1)$	$3.68 \cdot 10^{10}$	3,822 (67.2)	$3.89 \cdot 10^{9}$	3,640 (71.4)
96	$12 \cdot 8, (6 + 1)$	$3.58 \cdot 10^{10}$	3,648 (70.6)	$3.80 \cdot 10^{9}$	3,456 (75.0)
96	$16 \cdot 6, (9 + 1)$	$1.68 \cdot 10^{10}$	3,648 (70.6)	$1.79 \cdot 10^{9}$	3,456 (75.0)
98	$14 \cdot 7, (11 + 1)$	$7.30 \cdot 10^{9}$	3,528 (73.9)	$7.78 \cdot 10^{8}$	3,332 (78.6)
128	$16 \cdot 8, (6 + 1)$	$1.79 \cdot 10^{10}$	3,712 (70.6)	$1.93 \cdot 10^{9}$	3,456 (75.0)
130	$26 \cdot 5, (7 + 1)$	$2.15 \cdot 10^{10}$	3,900 (65.9)	$2.24 \cdot 10^{9}$	3,770 (70.0)
156	$26 \cdot 6, (9 + 1)$	$8.26 \cdot 10^{9}$	3,744 (70.6)	$9.07 \cdot 10^{8}$	3,432 (75.0)
160	$20 \cdot 8, (6 + 1)$	$1.80 \cdot 10^{10}$	3,680 (70.6)	$1.89 \cdot 10^{9}$	3,520 (75.0)
176	$22 \cdot 8, (6 + 1)$	$1.79 \cdot 10^{10}$	3,696 (70.6)	$1.89 \cdot 10^{9}$	3,520 (75.0)
182	$26 \cdot 7, (11 + 1)$	$3.54 \cdot 10^{9}$	3,640 (73.9)	$3.96 \cdot 10^{8}$	3,276 (78.6)
189	$21 \cdot 9, (7 + 1)$	$9.11 \cdot 10^{9}$	3,591 (73.2)	$9.68 \cdot 10^{8}$	3,402 (77.8)
190	$38 \cdot 5, (7 + 1)$	$2.10 \cdot 10^{10}$	3,990 (65.9)	$2.22 \cdot 10^{9}$	3,800 (70.0)
189	$21 \cdot 9, (7 + 1)$	$9.11 \cdot 10^{9}$	3,591 (73.2)	$9.68 \cdot 10^{8}$	3,402 (77.8)
190	$38 \cdot 5, (7 + 1)$	$2.10 \cdot 10^{10}$	3,990 (65.9)	$2.22 \cdot 10^{9}$	3,800 (70.0)
192	$24 \cdot 8, (6 + 1)$	$1.81 \cdot 10^{10}$	3,648 (70.6)	$1.92 \cdot 10^{9}$	3,456 (75.0)
189	$21 \cdot 9, (7 + 1)$	$9.11 \cdot 10^{9}$	3,591 (73.2)	$9.68 \cdot 10^{8}$	3,402 (77.8)
190	$38 \cdot 5, (7 + 1)$	$2.10 \cdot 10^{10}$	3,990 (65.9)	$2.22 \cdot 10^{9}$	3,800 (70.0)
192	$24 \cdot 8, (6 + 1)$	$1.81 \cdot 10^{10}$	3,648 (70.6)	$1.92 \cdot 10^{9}$	3,456 (75.0)

powered off. Once a parity disk's *Cache2Disk* task is started, it runs until all outstanding iPGs are persisted. To determine when to start them, we use the following simple policies for the evaluation:

1. An offline disk is started when it holds cached data for more than 180 seconds.

2. An offline disk is started when more than 5 GB of data is outstanding for it.

3. An online disk starts the *Cache2Disk* task when more than 512 MB of data is outstanding for it.

4. If the cache is 80% full, both online and offline disks with more than 512 MB of outstanding data start a *Cache2Disk* task.

Different factors like the RAID configuration, the current write-mode, the cache size, disk idle times, and the *Cache2Disk* execution frequency and policies define the behavior of the system. To achieve an energy-proportional throughput, a storage driver and a set of

Table 4.4: Feasible configurations for a 192 disks system. All disks use a fixed 16+1 SPC code for intra-disk redundancy.

ID	d	$\mathbf{x \cdot y, (hn + hk)}$	Eff.	MTTDL	#HPGs	Writes $\min(\lfloor \frac{x}{1+hk} \rfloor$, #HPGs)	Recovery $(hn + hk)/y$
1	$6 \cdot 32$	$8 \cdot 4, (5 + 1)$	58.8%	$3.76 \cdot 10^{12}$	4	$6 \cdot 4$	6 / 4
2	$3 \cdot 64$	$8 \cdot 8, (6 + 1)$	70.6%	$6.82 \cdot 10^{11}$	8	$3 \cdot 4$	7 / 8
3	$2 \cdot 91$	$13 \cdot 7, (5 + 1)$	67.2%	$7.73 \cdot 10^{11}$	13	$2 \cdot 6$	6 / 7
4	$2 \cdot 96$	$16 \cdot 6, (9 + 1)$	70.6%	$3.20 \cdot 10^{11}$	8	$2 \cdot 8$	10 / 6
5	$2 \cdot 96$	$12 \cdot 8, (6 + 1)$	70.6%	$6.81 \cdot 10^{11}$	12	$2 \cdot 6$	7 / 8
6	180	$36 \cdot 5, (15 + 1)$	70.6%	$5.16 \cdot 10^{10}$	9	9	16 / 5
7	189	$21 \cdot 9, (7 + 1)$	73.2	$1.73 \cdot 10^{11}$	21	10	8 / 9
8	190	$38 \cdot 5, (7 + 1)$	65.9%	$4.40 \cdot 10^{11}$	19	19	8 / 5
9	192	$24 \cdot 8, (13 + 1$	76.5	$6.86 \cdot 10^{10}$	12	12	14 / 8
10	192	$24 \cdot 8, (6 + 1)$	70.6	$3.44 \cdot 10^{11}$	24	12	7 / 8

placement strategies are required that coordinate how data is written to the RAID. To better understand the different system behaviors and to lay the foundation for such a storage driver, we created a set of micro benchmarks that trigger specific bad or sweet spots of the system.

4.4.5 Simulated Disk Backend

To speed up the evaluation and to be able to test many different combinations, we developed a mock back-end for the LoneStar RAID. The physical disks are replaced by mock disks that can only store iPGs that contain either zeros or ones. They can be efficiently represented in main memory and their actual throughput can be configured. When an iPG is written to the cache or a disk, its content is checked and mapped to a single bit in a vector. When the cache or disk is read for a specific offset, the vector is checked and an iPG of zeros or ones is created and returned.

To model a realistic system behavior, we induce latencies for disk and cache accesses and for disk spin-ups. We serialize the access to the cache and the individual disks and let the system sleep for some time to model the hardware access. To model a cache with a total throughput of 800 MB/s we add a latency of 83 μs and a disk capable of reading or writing 100 MB/s, each iPG operation adds up 664 μs. To model the mechanical parts of the disk drive, an additional seek time of 9 ms is added before each access to a consecutive range of iPGs. The visualization of sequential read requests in Figure 4.8 shows that for 7 disks on average 8,000 iPGs are read per second which for 68 kB per iPG results in 76 MB/s per disk. The 24 MB/s gap to the theoretical speed can be explained by a small XOR overhead as the iPGs are checked on read and overheads and inaccuracies of internal clocks and timers of sleeping threads to simulate the disk access times. The evaluation focuses on micro-benchmarking to foster the understanding of the RAID scheme behavior. Therefore, we

chose a small idle time of 20 seconds after which the disk is powered off. When a sleeping disk's data is requested, we induce a 5 second delay for the spin up.

By reducing the test environment to this back-end, we are able to test the performance of the software prototype and study the impact of different configurations of the LoneStar RAID. Omitting the disk drives' and caches' capabilities, the tests' results actually depend on the remaining hardware performance as the journal is persisted and all data is encoded and processed.

We implemented a test-client that loads an instance of the LoneStar RAID with a particular configuration. For each test run 64 threads are started that work off a list of generated operations. To be able to compare the results, we fixed the amount of data written by the clients. In total, 32 GB of data was read or written, resulting in 524,288 iPGs of 68 kB that were processed by the LoneStar RAID core. For all tests, the cache was big enough to store all data, except for the test run presented in Figure 4.12, where the cache was restricted to 16 GB to analyze a cache overflow scenario.

We used the configurations in Table 4.4 and modeled workloads that either hit all data disks of one vertical parity group (SingleVPG), all data disks of a horizontal parity group (SingleHPG), or one data disk of each horizontal parity group so that all $\lfloor \frac{x}{2} \rfloor$ vertical parity disks are active (AllVPDisks). The latter set of disks hit all independent disk groups as presented in Figure 4.2 and theoretically yields the best possible performance for direct writes. The SingleVPG and SingleHPG disk sets on the other hand can benefit from caching.

We measured the direct and delayed write modes discussed in Section 4.3.5. The workload generator created new operations for the 64 client threads. Each operation is either a read or write and has a target bucket and a target range. We varied the sizes of the operations to 20% with 1 to 4 iPGs and 80% of large operations of 50 to 100 iPGs.

To test the caches capabilities to merge data as depicted in Figures 4.5a and 4.5b, we investigated two different write patterns. The random distribution mode (Rand) chooses a random bucket of the target disk set, and a random range, which minimizes possible blocking of locks but also the efficiency of merged caches. In the sequential mode (Seq), all buckets of the target disk set are filled in a sequential way starting at offset 0, so that on average all buckets contain the same amount of data on the same range. For 10 buckets of an HPG and a total of 1000 iPGS written, on average each bucket's first 100 iPGs would be written in this way which results in a merged 100 iPGs for the horizontal parity disk.

The main metrics of the evaluation are the number of read or written iPGs on the testing application, which marks the *effective throughput* of the system. Analog to that, the total number of read or written iPGs actually processed by the disks is called *internal work*.

Furthermore, we are interested in the number of disk spin-ups and the disks' total runtime as they define the energy consumption of the system. The main metric to argue about the efficiency of the LoneStar system is the ratio of effective throughput to the energy consumption related to the internal work. We focus on the disks' power consumptions only and use a simplified metric of effective throughput to the sum of all disks' runtimes. With

a concrete model of disk drive power characteristics, this metric can be directly transferred to $\frac{MB}{Joule}$.

4.4.6 Simulation Results

In the following we will present multiple figures that visualize the micro benchmarks discussed in the previous section. They are based on the test runs of a $x = 21, y = 9, hn = 7, hk = 1$ configuration and visualize the actual work on the disks, their runtime and the amount of cached iPGs. The upper graph presents a 5 second rolling mean of the system's internal work. The stacked areas present reads (r) and writes (w) that hit the data (d), vertical (vP) and horizontal (hP) parity disks. The lower graphs show the number of currently cached iPGs and the number of running data and parity disks. Each figure visualizes a single test run. All simulations were run on a server containing an Intel Xeon E3-1230 V2 CPU with 3.30GHz, 16 GB Ram, and an Intel SSD 520 Series SSD with a default ext4 file system. The operating system was an Ubuntu 12.04.3 64bit.

The test run presented in Figure 4.8 reads data from all data disks of a horizontal parity group. All 7 disks are running and about 8,000 iPGs/s are processed. Given the 64 kB data portion of an iPG, this results in an effective throughput of about 500 MB/s.

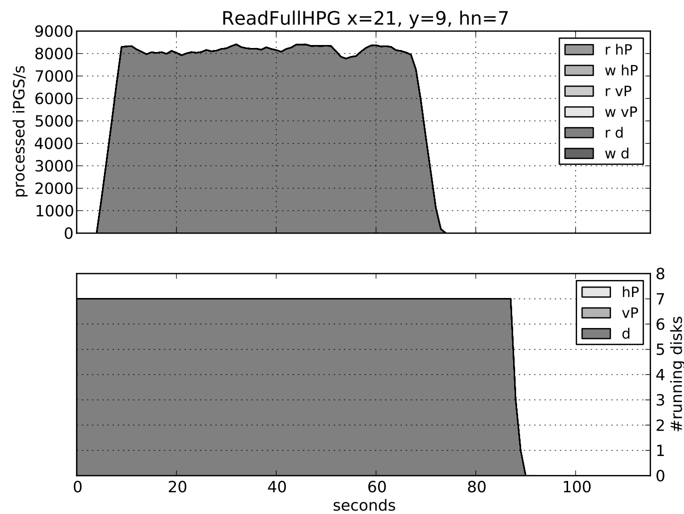


Figure 4.8: Reads to all disks of a HPG.

The *AllVPDisks Direct* presents the theoretically highest direct write throughput. Here, the maximum number of data disks that do not share any parity disks, are written in parallel. The lower part of Figure 4.9 shows the 10 running data disks alongside the accompanying 20 vertical and 10 horizontal parity disks. The upper part of the figure shows the summed up 40,000 processed iPGs/s of internal work for the bottom 5,000 *w d* iPGs/s

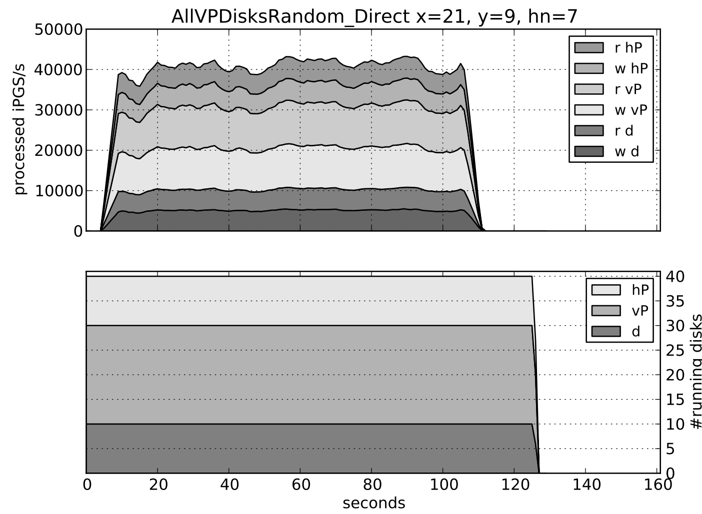


Figure 4.9: Direct writes to random ranges on non-overlapping parity groups.

that show the actual effective throughput. These figures highlight the overhead factor of 8 for writes.

In contrast to these direct writes to independent disks, Figure 4.10 shows how the *hP* disk is the bottleneck for direct writes to all data disks of the horizontal parity group. Every write to any data disk results in a RMW cycle on the single horizontal parity disk. Although 7 data disks are running, the effective write throughput over all data disks 'w d' is capped at the speed of the shared parity disk. In addition to this, the test used the sequential write mode where all 64 threads sequentially fill the data disks. As all disks are sequentially filled starting at offset 0, tasks will inevitably lock each other, which explains the high variance in the throughput.

The test runs visualized in Figures 4.11 and 4.12 investigate the caching behavior of delayed random writes to all disks of a horizontal parity group. All tests write a total of 32 GB of data to the RAID and we chose a cache of 64 GB that can hold all outstanding iPGs. Only for the visualized test in Figure 4.12 we restricted the cache to 16 GB to test an overflow behavior. Both tests use the data disks only and start to fill up the cache. With about 4,000 written iPGs/s after about 20 seconds of work, the *WatchDog* task triggered the 5 GB outstanding iPGs threshold of the *hP* and *vhP* disks. Therefore, the parity disks were started to run a *Cache2Disk* task. When the test application finished all write operations it executed a *SyncRAID* which starts a *Cache2Disk* task on all parity disks with outstanding iPGs and blocks until all data is fully persisted. This behavior can be seen in Figure 4.11 where both the number of cached iPGs and the number of running disks climax. All writes to the data disks finished and they are shut off after 20 seconds of idling. At the same time the other vertical parity disks are started.

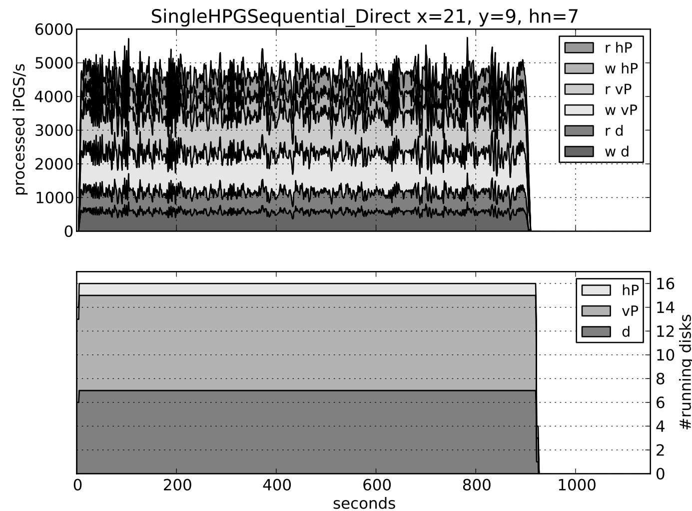


Figure 4.10: Direct writes to sequential ranges to all buckets of a HPG.

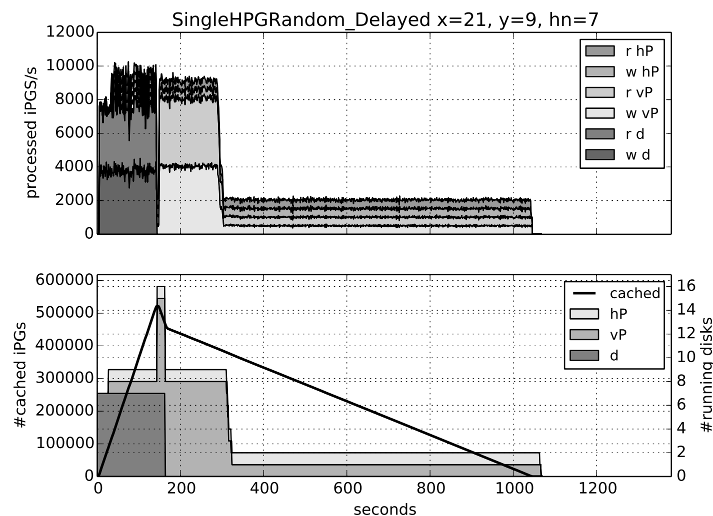


Figure 4.11: Delayed writes to random ranges to all buckets of a HPG.

Another behavior can be observed in Figure 4.12 when the cache is filled up. According to the proposed *WatchDog* policies, all vertical parity disks are started as they each have more outstanding data than the 512 MB threshold. At the same time the write rate to the data disks drops as the creation of a new delayed write promise is blocked until its requested size can be reserved on the cache. Therefore, data writes are limited to the speed of the *Cache2Disk* tasks. Once started, the *Cache2Disk* task runs until no outstanding data

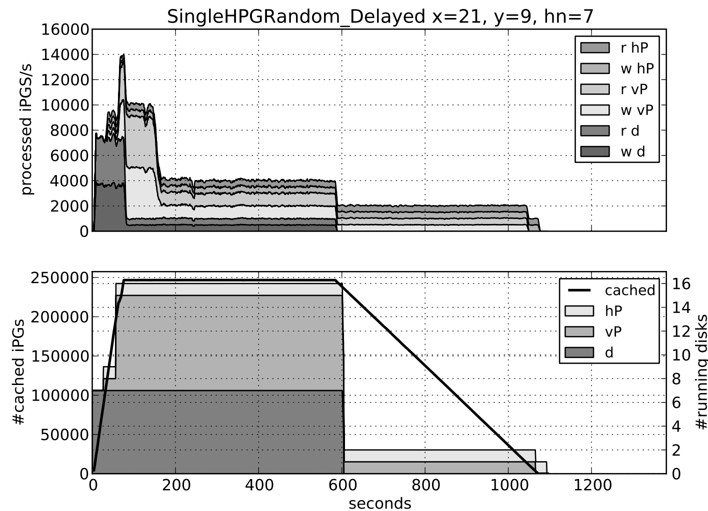


Figure 4.12: Delayed writes to random ranges to all buckets of a HPG. 32 GB written with 16 GB cache.

is left. This explains the constant work of the vertical parity disks until the data writes finish, as new data is constantly cached and the *Cache2Disk* never runs out of work.

Both tests show a similar behavior after all data was persisted to the data disks and their vertical parity disks. Because the test wrote to random ranges to the data disks resulting in non-mergable cached data, the hP and vhP disks need to write the full 32 GB persisted data. Therefore, these disks still run for some time working off the cached iPGs.

A different caching behavior can be observed in Figure 4.13 that shows sequential delayed writes to all disks of a horizontal parity group. While the initial behavior is similar to the random writes of the previous tests, the syncing of the RAID is much faster as all data can be merged. The merging results in the same number of processed iPGs on all involved disks, which can be seen by the simultaneous finishing of writes for the parity disks. While the random workloads cannot merge any data, each outstanding promise is individually finished in sequence, which results in the linearly falling cached iPGs line. For the merged tasks on the other hand, multiple promises are merged and can be removed from the cache as a compound operation. This results in the stepped degradation of the line.

Like the sequential direct workload shown in Figure 4.9, the initial write throughput of this test shows a high variance. Again, as all disks are filled starting from the same offset, threads will block each other. This becomes especially visible for the first seconds of the test until the effective throughput reaches a plateau above 2,000 iPGs/s.

As discussed in Section 4.3.4 the ratio of effective throughput to internal work can be reduced by caching. Given the 1:8 baseline for direct writes, for a single horizontal or vertical parity group we observed factors close to the theoretical optimum. In the evaluation

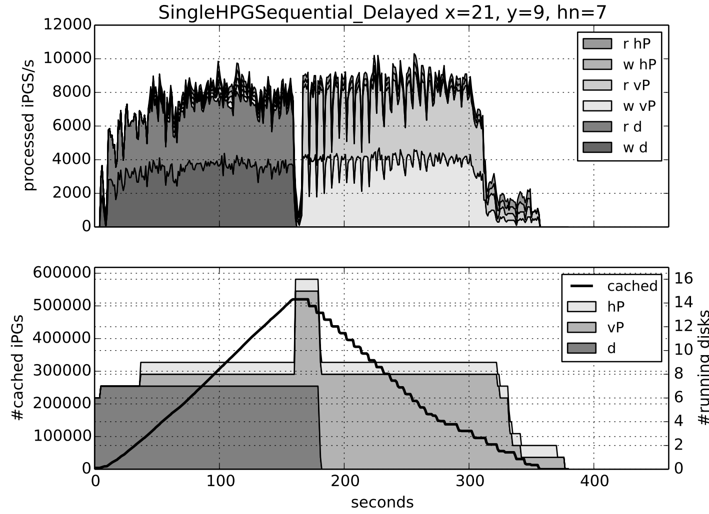


Figure 4.13: Delayed writes to sequential ranges to all buckets of a HPG.

we found a minimum of 1:4.31 for the $x = 36, y = 5, hn = 15, hk = 1$ (ID 6), which is close to the theoretical optimum for that configuration of 1:4.27.

Table 4.5: Ratio of effective throughput to sum of all disks' runtimes. Metric is equivalent to $\frac{MB}{Joule}$.

ID	$\mathbf{x \cdot y, (hn + hk)}$	**FullHPG**	**AllVPDisksSeq**			
		ReadSeq	**Rand**		**Seq**	
			Direct	**Delayed**	**Direct**	**Delayed**
1	$8 \cdot 4, (5 + 1)$	925.3	125.7	112.4	125.4	119.2
2	$8 \cdot 8, (6 + 1)$	879.8	125.3	112.4	125.6	119.3
3	$13 \cdot 7, (5 + 1)$	927.3	117.3	101.3	117.3	107.5
4	$16 \cdot 6, (9 + 1)$	766.9	109.6	82.9	109.8	87.4
5	$12 \cdot 8, (6 + 1)$	881.9	116.8	101.8	117.1	107.3
6	$36 \cdot 5, (15 + 1)$	605.0	105.9	74.8	106.1	79.8
7	$21 \cdot 9, (7 + 1)$	841.4	102.4	68.9	102.3	73.4
8	$38 \cdot 5, (7 + 1)$	839.2	74.4	37.6	74.8	40.8
9	$24 \cdot 8, (13 + 1$	653.6	96.2	58.8	96.2	63.0
10	$24 \cdot 8, (6 + 1)$	877.0	96.1	58.7	95.8	63.0

In Section 4.4.5 the ratio of effective throughput to the disks' runtimes was defined as a metric to compare the efficiency of different configurations and workloads of the micro benchmarks. Tables 4.5 and 4.6 present these ratios for the aforementioned workloads for the configurations defined in Table 4.4. For every configuration we tested the different sets of target buckets *SingleVPG*, *SingleHPG*, and *AllVPDisks* with random and sequential write patterns with direct and delayed write modes. We also present the numbers for reading 32 GB of test data from all buckets of a HPG. Every test was executed for at least 20 times

Table 4.6: Ratio of effective throughput to sum of all disks' runtimes. Metric is equivalent to $\frac{MB}{Joule}$.

ID	$\mathbf{x \cdot y, (hn + hk)}$	**SingleVPG**				**SingleHPG**			
		Rand		**Seq**		**Rand**		**Seq**	
		Direct	**Delayed**	**Direct**	**Delayed**	**Direct**	**Delayed**	**Direct**	**Delayed**
1	$8 \cdot 4, (5 + 1)$	56.6	121.6	56.4	145.7	47.1	119.9	46.9	173.1
2	$8 \cdot 8, (6 + 1)$	29.8	111.9	29.6	131.8	40.4	118.1	40.2	169.2
3	$13 \cdot 7, (5 + 1)$	35.4	114.8	35.2	136.1	47.1	120.0	46.9	173.3
4	$16 \cdot 6, (9 + 1)$	35.4	114.6	35.2	135.7	28.3	112.5	28.1	149.3
5	$12 \cdot 8, (6 + 1)$	29.8	112.0	29.7	132.3	40.4	118.0	40.2	170.4
6	$36 \cdot 5, (15 + 1)$	43.6	118.1	43.4	141.3	17.7	95.8	17.3	110.1
7	$21 \cdot 9, (7 + 1)$	25.7	108.7	25.6	124.5	35.4	116.5	35.2	163.7
8	$38 \cdot 5, (7 + 1)$	43.5	118.3	43.4	141.1	35.4	116.5	35.2	164.2
9	$24 \cdot 8, (13 + 1$	25.8	108.9	25.6	124.5	20.2	102.1	19.9	123.0
10	$24 \cdot 8, (6 + 1)$	29.8	112.0	29.7	131.6	40.4	118.1	40.2	169.2

and we calculated their means and 95 % confidence intervals. The tests were repeated until the confidence intervals fit into the mean's surrounding 1% interval.

The results are normalized to the amount of processed data. Nevertheless, the different sets of target buckets and the different configurations with different numbers of parity groups and disks within these groups need to be considered. For every disk spin-up five seconds of starting and 20 seconds of idling before being shut off are counted into total disks' runtime. For large amounts of data written to a small number of disks, this overhead is much higher than a small amount of data spread to a high number of disks. Therefore, the presented results can only be used to identify and argue about tendencies between the different configurations.

The analysis of the workloads underline the assumptions of Section 4.3.7. As expected, the *FullHPG* read tests show the highest efficiency compared to the writing workloads. The comparison of the different configurations for the *ReadSeq* workload shows the best values for the smallest number of used disks (hn) and the worst result for ID 6 (see Table 4.6) which runs 15 disks. Here, the aforementioned start and idling overhead of the disks becomes visible. Nevertheless, the factors close 8 relative to direct writes and factors near 5 relative to sequential delayed writes can be observed.

The outcomes for *AllVPDisks*, presented in Table 4.5, show the overall best results for direct writes as no locks can block, no parity disks are shared, and all involved disks can operate at full speed. Correlating the numbers of possible parallel writes from Table 4.4, the results get worse with raising number of parallel writes. For configuration 8, for example, a total of 19 data disks are written in parallel. With the parity disks this test used a total of 76 disks to persist 32 GB of data. Setting the summed up 5 seconds startup times for the 76 disks in relation to the amount of work that the disks actually processed, the

aforementioned overhead becomes visible. Further observations for the *AllVPDisks* show that random and sequential direct workloads have a similar speed and that sequential delayed results are slightly better than the random delayed as the *Cache2Disk* can operate on larger sequential blocks. The tests show that the delayed write mode is more expensive. Especially with rising number of disks that are involved, the ratio declines. This can be explained by the current implementation of the *Cache2Disk* task. Per disk only a single task is running that works off data sequentially. First it reads the outstanding promises from the journal and works them off in a sequential manner. Mergeable promises are consolidated and updated in the journal. The old data is read from the parity disks, the promises' data is read from the cache and the new data is calculated and written back to disk. In contrast to that, the direct write test applications use 64 client threads in parallel that work on the set of disks which can fully utilize the disk and CPU.

Table 4.6 presents the results for tests that use all disks of a vertical or horizontal parity group. In general, when multiple disks of a parity group are written, the delayed write mode always improves the efficiency of the system. Especially for sequential workloads, as less work hits the disks. For the direct workloads that use disks of the same horizontal or vertical parity group, with rising number of disks, the efficiency declines, which can be explained by the shared parity disk bottleneck. On Average, for direct workloads, the sequential write patterns yield a slightly lower efficiency due to the locking overhead.

A comparison of workloads that hit the same number of data disks like a VPG of ID 10 and an HPG of ID 2, which both use 6 data disks, shows a difference of 29.8 to 40.4. Recapitulating the assumptions of Section 4.3.7, this can be explained by the higher number of involved parity disks. While writing the six data disks of the HPG uses 8 parity disks, writing to the VPG utilizes 12 additional disks.

4.5 Discussion & Conclusion

We presented a new two-dimensional RAID scheme that uses up to hundreds of disk drives and integrates journaling and caching concepts to improve energy-proportionality of IO operations, throughput and reliability. The evaluation analyzed the elastic fault tolerance of the scheme and a micro-benchmark suite investigated the system behavior under various conditions. The results of the benchmarks and the discussions in this article showed that the LoneStar RAID has a huge potential to save energy by delaying and merging operations and by aggressively powering off idle disks. Nevertheless, the correct usage of the system has a strong impact on the performance and disk energy consumption.

The main design goals that led to the proposed architecture of LoneStar were the usage of many disks that are mostly powered off, and as little surrounding hardware as possible to drive the system. Many different codes, RAID schemes, and hardware designs were previously proposed. Therefore, we discuss and compare some competitors to classify LoneStar.

LoneStar uses intra-disk and inter-disk redundancies and was designed for small CPUs that control many disks. During the design phase, the calculation of strong erasure codes was prohibitively expensive and would have required multiple strong CPUs. For this reason, our system solely relied on cheap XOR operations achieving a coding throughput of multiple gigabytes per second [GSB$^+$11]. Today, new advancements in erasure coding provide coding schemes that achieve both, a high reliability and a throughput that is now suitable for storage system applications [PGM13].

The used inter-disk redundancy scheme and the distribution strategy define the performance, flexibility and energy efficiency of the system. A strong $n + k$ erasure code like $12 + 4$ reveals a great storage efficiency and MTTDL (see Table 4.2). In combination with a huge sea of hundreds or thousands of disk drives and a declustering distribution strategy that spreads the file parts over all available disks, very reliable storage can be build. The declustering reduces the rebuild times, which strongly influence the MTTDL of the system. The downside of such a system is that all disks need to be powered at all times. With a fixed mapping of data and parity disks for a $n + k$ code, which does not use declustering, the number of active disks can be reduced. For example, if such a cluster is fully written, the parity disks can be spun down until a recovery is required. The main disadvantage of LoneStar and classical clustered RAID approaches are the high rebuild times of failed disk drives. Especially with growing disk capacities and disk throughputs that hardly improve, the rebuild times will further rise. On the other hand, the rebuild architecture of LoneStar and its elastic fault tolerance make it robust against data loss due to failed rebuilds.

The erasure code works well for full stripe writes that fill the n data blocks. An in-place update of any of the n data blocks, on the other hand, is expensive in terms of IO operations and computation, as all other $n - 1$ blocks need to be read again to re-calculate and write the k parity blocks. Due to its intertwined parity groups, LoneStar does not provide full stripe writes and each write to the system requires a full RMW-cycle. On the other hand, in-place updates are possible with well-defined consistency guarantees and only require a total of 4 disk drives. LoneStar uses 64 kB as a stripe unit. Thus, updating a single bit, requires to read 4 x 64 kB from four disks and to write the total of 256 kB back to the same disks again. An update within a $12 + 4$ code with the same stripe unit size would read 704 kB and write 256 kB and use a total of 16 disks. Today's cloud storage architectures therefore use erasure codes on tiers where data is not supposed to change [HSX$^+$12a, SAP$^+$13].

When building and maintaining a large-scale archive, the costs for hardware and energy have to be considered. The related work in Section 4.2 presented Pergamum [SGMV08] and Pelican [BBD$^+$14] that render the extremes of disks per CPU. Pergamum uses a small NVRam and a CPU per disk, whereas Pelican houses 1,152 disks with 2 servers per rack. Both systems are designed for disk-based cold archives, with the primary goal of keeping disks shut-off and limiting the number of concurrently active disks. While Pergamum also provides data access within the seconds of disk spin-up, Pelican batches and schedules requests as powering up sets of disks depends on conflict domains like cooling or reliability.

Therefore, the time to first byte can be multiple hundreds of seconds which renders the system unusable for active environments. Pergamum, Pelican, and the LoneStar RAID have in common that the actual data placement strategies are delegated to the accessing client applications. Therefore, it is hard to argue about and compare the resulting system performance. The same holds for the evaluation of the reliability of the systems. Both, Pelican and Pergamum are evaluated with a $n + 3$ erasure code, but can be configured to use stronger codes. Pergamum presented a configuration that would provide a MTTDL of $1.25 \cdot 10^7$ hours, or about 1,400 years for 10 PB of user data. A LoneStar configuration with 64 4 TB disks per server would result in 57 machines and a MTTDL of $3.59 \cdot 10^{10}$ hours, or about 4 million years.

Though being designed for archival workloads, LoneStar provides a huge flexibility as it can be used for cold archives and scenarios where data is accessed regularly. While reads are efficient and simple, efficiently writing data to the system requires some planning. First, a good placement strategy that maps data to buckets is required, as it also has a strong influence on the read performance. In the best case, multiple consecutive read requests from an application hit a small number of buckets, but in the worst case, every read operation needs to spin up another disk. Writes are fast as long as the cache is not full. Multiple dependent data disks can be updated in parallel without interfering parity disk updates. But once the cache is filled, the write throughput is defined by the shared parity disks. In the worst case, all written buckets share a parity disk that then defines the total throughput.

Systems with defined write workloads, like data ingests in archival environments or move operations from a hierarchical storage manager, can alleviate the bottlenecks of the LoneStar RAID by providing a storage driver that aggregates operations and places data to fitting buckets. Subsequent work on the LoneStar concept led to the LoneStar Stack, which implements a POSIX-compatible file system using the LoneStar RAID for data storage [GBSB14]. The system moves the caching to the file system layer and uses serialized direct writes as proposed in Section 4.3.5. Based on file system metadata, a placement strategy gathers new and related files and decides their mapping to the data buckets. A background process monitors the disks' states and the cache's fill level to flush data to RAID as efficiently as possible. A good strategy to place related data to the same buckets also has a huge potential for energy savings. The study of the ECMWF storage landscape reveals many domain-specific placement hints to be available [GNM+15]. Their archive for numerical weather data, for example, can be well-partitioned by domain-specific contexts like tiles, years, or countries. New data is constantly added that can be either added to existing context specific buckets, or appended to buckets based on its creation date. Even multiple copies of the same data can be created in related buckets to improve the robustness and chances to spin up less disks as related data is clustered. A distribution of data to the buckets that is tailored to common request characteristics of the domain-specific query language could reveal huge energy savings. The system could also be used in the

Internet Archive scenario, or any other active archive scenario, where relations between data or access patterns are apparent and can be encoded into a distribution strategy.

In conclusion, the presented RAID scheme is well-suited for large archives where data is written once, rarely updated, and accessed sometimes. Though it can be used to build cold archives, it excels in scenarios where related data is read in irregular intervals. Especially, if data is ingested by a storage manager that features semantic data placement strategies, LoneStar provides high read throughput, high reliability, low energy consumption, and data is available within seconds.

5 LoneStar Stack: Architecture of a Disk-Based Archival System

Abstract

The need for huge storage systems rises with the ever growing creation of data. With growing capacities and shrinking prices, "write once read sometimes" workloads become more common. New data is constantly added, rarely updated or deleted, and every stored byte might be read at any time - a common pattern for digital archives or big data scenarios.

We present the LoneStar Stack, a disk based archival storage system building block that is optimized for high reliability and energy efficiency. It provides a POSIX file system interface that uses flash based storage for write-offloading and metadata and the disk-based LoneStar RAID for user data storage. The RAID attempts to spin down disks as soon and as long as possible. For reads, only a single disk is accessed, while writes require 3 additional parity disks to be spun up. The cache aggregates new files and a semantic data placement engine decides how they are persisted to the RAID. Asynchronous data movers then persist the data.

The system provides an end-to-end data integrity, an elastic fault tolerance that can at least recover from all 3-disk failures, and provides multiple paths for data integrity checking and recovery. The system can use 70% of the raw disk capacity and is optimized for fast reads with a minimum number of powered on disk drives.

5.1 Introduction

The need for huge storage archives rises with the ever growing creation of data. Next to tape, hard disk drive based storage systems became a considerable alternative for cheap mass storage. Compared to tape, disks have very different reliability, performance, and energy characteristics that need to be tackled to create an efficient archival system.

In this paper we focus on storage usage scenarios like large media archives or digital public libraries, where all data has to be online at any time. Documents are constantly added, rarely deleted and never updated in place. Reads occur sometimes to retrieve single documents or collections of documents - a "write once read sometimes" workload.

A tape based solution is reliable and cheap, but suffers from high latencies. Furthermore, the number of parallel accesses is limited by the number of available drives. A more responsive solution can be build with disk drives. Leventhal argues that for today's storage systems classical RAID6 systems with double parity are no longer resilient enough [Lev09], as data volumes of single systems rise and the probability for a data loss becomes a severe threat. Disk based object stores like OpenStack Storage or Amazon S3 are another viable solution. Typically, they use triplification [WBM+06] or erasure coding [HSX+12b] and randomized distribution strategies. The advantages of high throughput and reliability have to be considered against their downsides, like a great number of spinning disks as data is encoded and distributed or striped over many disks. To access even a single bit of information, multiple disk drives are required.

Considering the costs of a large data archive, energy consumption also receives elevated interest. In contrast to tapes that do not require any power while not in use, an idle disk drive still uses a small amount of energy. Furthermore, every access to an idle disk requires an expensive spin-up. To achieve a high reliability, multiple replicas of a file can be stored, which results in additional storage capacity that has to be bought and powered. Modern cloud based storage systems store a single copy of a file and use erasure coding to be able to compensate the loss of multiple disks with a low storage overhead of 1.33x [HSX+12b, SAP+13]. Data is erasure encoded and spread to 16 blocks, where any 12 of them are required to decode the data object. To achieve the best reliability, these blocks have to be spread to 16 different error domains, like independent disks. Here, a read would require 12 and a write 16 spinning disks.

In this paper we present the architecture for an energy-efficient and highly reliable disk-based storage solution that is tailored to the aforementioned "write once read sometimes" workloads. It is designed for a single server that hosts a reliable NVRAM and up to 240 disk drives. We use an improved version of the previously proposed LoneStar RAID [GMPB11] and now contribute the new LoneStar FS to present a fully integrated storage stack. The system yields end-to-end data integrity with multiple paths of validation and recovery and has an elastic fault tolerance that is more than 3-disk failures tolerant. Furthermore, at every point in time the storage system is in a defined state that is persisted to the NVRAM or disks. Therefore, the system is also secure from data loss due to power outages.

We assume that all disks are powered off by default. To read a file, only one target disk has to be spun up, while writing new files requires three additional parity disks to run. During reads, the intra-disk redundancy mechanism detects and corrects disk errors. Additionally, the file system stores checksums for file contents and the individual file's extents. New files are first written to the NVRAM. The semantic data placement engine uses file metadata or hints about file contents to decide to which target disk each file should be persisted. In this way related files or collections of files can be placed on the same set of disks. If there is enough outstanding data for a data disk, so that it is worthwhile to spin up disks for writing, or the cache fill level is critical, data is moved from the cache to the LoneStar RAID.

The main contribution of this paper is the new LoneStar FS layer with end-to-end integrity and semantic data placement strategies to implement an energy efficient hierarchical storage management. The file system targets and overcomes the shortcomings of the LoneStar RAID architecture to build a full storage stack for a highly reliable and energy efficient archival system. We discuss the architecture and evaluate the proposed chain of reliability features and evaluate our implementation on a 72-disk based server.

5.2 Related Work

The main question of how to reliably store huge amounts of data to disk drives as well as multi-tiered architectures that use flash media has been subject to research for some time now.

The architecture of the internet archive as described by Jaffe and Kirkpatrick [JK09], for example, is a disk-based system serving more than a petabyte of online data.

Today, flash-disk hybrids are found in computer systems ranging from consumer laptops to high performance storage solutions. Apple's Fusion Drive, for example, works on the file system layer and keeps frequently used files on flash, while other systems transparently add a flash based caching layer on top of a block device at the kernel level [fbf]. Another approach are new hybrid disk drives that internally use a small flash based write-back cache. Two common goals of using a multi-tier storage architecture are the improvement of performance while also reducing the energy consumption [KV08, BBL06, AvMT12, SKM12].

Narayanan et al. analyzed block level traces of enterprise datacenter workloads and found disk idle times that are worth to be exploited [NDR08]. They proposed the usage of an SSD to offload writes to spun down disks. These are applied to the disk later without sacrificing consistency or failure resilience. They were able to show up to 60% energy savings with disk spin downs and write-offloading.

Felter et al. presented a reliability-aware hybrid storage system that combines power-aware flash caching and disk spindown [FHC11]. To limit the rate of disks being powered on and off, they use a token system that limits the power cycles and reduces hardware wearout. To reduce the IOs on the disks, they introduce an SSD as a cache for the disks.

Guerra et al. also analyze a multi-tiered system that uses flash and disks. They evaluate different strategies of when to use which tier and how to efficiently migrate data [GPG+11].

Kaiser et al. proposed the ESB system, which extends an ext2 file systems and stores metadata blocks to an SSD while data is stored to the disk [KMHB12].

Most of the commonly used file systems, like ext4, lack any support for data or metadata checksumming. Therefore, every hardware error will result in inconsistent data. Prabhakaran et al. analyzed multiple file systems for their capabilities to handle common disk errors, like entire disk failures, block failures, or block corruption. Based on their findings they proposed ixt3, which improves ext3 with data and metadata checksumming, metadata replication, and parity based redundancy for user data.

The ZFS file system that was thoroughly analyzed by Zhang et al. [ZRADAD10] provides a full end-to-end data integrity and incorporates different RAID strategies to spread data to multiple disks. ZFS also supports a flash based write-back cache, which is primarily used to conceal the shortcomings of the disk based backend.

The use of disk drives for storage archives was proposed by Colarelli and Grunwald [CG02]. Furthermore, various options to improve the energy consumption of disk based storage systems were proposed [LW04, WZL08, WOQ+07, XYAZ11, GMPB11].

Greenan et al. explored how to apply erasure coding to spin down disks [GLM+08]. The Pergamum system proposed by Storer et al. [SGMV08] also uses erasure coding and algebraic signatures to spread data over multiple autonomous storage tombs to build scaling out disk based archival stores.

Wildani and Miller explored semantic data placement strategies to group and aggregate related data to be stored on the same storage target. They show that a good placement strategy can reduce the number of disk spin ups in archival stores [WM10].

5.3 LonesStar Storage Stack

Today's digital archives grow in the available capacity as well as in the number of read requests. The LoneStar Stack aims at overcoming the bottlenecks of tape based archives for regular random read requests over the full archive while being highly reliable and energy efficient. We propose a building block of a single server that can be used to build scale out archives. Each server is comprised of a huge number of disk drives that are assembled into the two-dimensional RAID. Additionaly, an NVRAM is used to persist a POSIX compatible file system layer that integrates a write-back cache to aggregate and combine writes to RAID.

The main challenge for the design of such a system is to find a balance between energy-consumption, reliability and performance. To optimize such a system, the disk drives' reliability, performance and energy models, in particular, need to be considered. A sleeping disk saves energy, but spinning it up takes some seconds and wears out the disk mechanics. Storing multiple copies of a file is reliable and improves the access performance

but requires additional capacity and energy to store and maintain the copies. The LoneStar RAID exports individual data disks, so that a full file can be stored on a single disk only. All disks are spun down as soon as possible so that read requests tend to spin up disks again. In the best case, files that are read in batch reside on the same disk; however in the worst case a disk is spun up for every file. Hence, the placement of files to disks becomes a crucial part of the efficiency of the storage system.

The target system is comprised of many components that are accessed and can fail in parallel. Therefore, mechanisms must be introduced to coordinate parallel accesses and maintain an always consistent state of the system.

In the following, we present a short summary of the LoneStar RAID, give a detailed overview of the LoneStar FS, introduce the infrastructure for semantic data placement strategies, delayed operations, and system management and discuss how we achieve consistent operations and end-to-end integrity.

5.3.1 LoneStar RAID

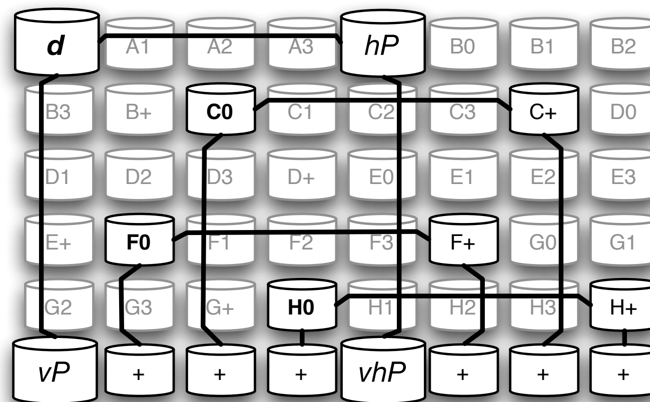


Figure 5.1: Schematic of an 48 disk LoneStar RAID. X0-X3 denote the buckets with their X+ parity disks of the horizontal parity groups. The bottom row contains the parity disks of the vertical parity groups.

The LoneStar RAID is a software layer that uses disk-based block devices for storage [GMPB11] and minimizes the amount of active disks for reliable read and write operations. Disks are arranged into a two-dimensional array so that every disk is part of two RAID4 schemes - a horizontal and a vertical parity group. The system uses dedicated data and parity disks and exports the data disks as individually accessible *buckets*. Hence, a single spinning disk is sufficient to read data off a bucket, depicted as *d* in Figure 5.1, while updates involve three additional parity disks, *vP*, *hP*, and *vhP*. Every disk is additionally

protected against block failures and latent sector errors by intra-disk redundancy (IDR). For every 16 data blocks of 4 kB, an additional 4 kB parity block is stored for checksumming and recovering of bad sectors [GSB+11]. If the disk's IDR scheme fails to recover a read error or the disk cannot be read at all, the disk can be recovered by either spinning up its horizontal or vertical parity group. This scheme achieves an elastic fault tolerance that can recover all 3-disk failures and, with decreasing chances, most of simultaneous 4, 5, ... drive failures, depending on the number of disks and the RAID configuration. Alongside the inherent intra-disk-redundancy, this RAID scheme achieves an MTTDL which is orders of magnitudes higher than RAID 6 based systems with the same number of disks and yields a comparable and competitive storage overhead.

The system performs well for reads and recovery operations, but write operations need to be handled with care. Due to the intertwined parity groups, the parity disks are shared by multiple groups. Furthermore, every write requires a full read-modify-write cycle on the data and the according three parity disks. If the writes to the data and parity disks occur in parallel, an error like a power outage would leave the RAID in an inconsistent and irrecoverable state. Every write involves 4 disks. If the exact state of a write is not known after an error, all 4 disks need to be recovered. Unfortunately, these exact 4 dependent disks cannot be recovered as they are contained in parity groups that depend on each other. This condition is called a RAID write hole and is an irrecoverable condition for the LoneStar RAID.

To maintain a consistent system state, even in case of power outages, we use a transaction based interface to access the block storage and keep a journal of started and finished operations. A further contribution of this paper is to serialize the actual write requests and mark their start and finish times in the journal. If a write does not finish, the journal helps to identify which disks have completed their write of the new data. The write can then be completed for the other disks. The disk order of the serialized writes is constant. If, for example, the new data was written to the d and hP disks and during the update of vP an error occurred, the written range is marked as erroneous on hP and vhP and a recovery process is started.

The serialization overhead alongside the read-modify write cycles on the disks result in a decreased write throughput for a single operation. Writes are divided into 4 MB chunks and multiple write operations can be active in parallel. With multiple parallel file writes that each split their work into smaller operations, the disks' bandwidths can be fully utilized.

5.3.2 LoneStar FS

The main contribution of this paper is a file system like middleware that provides a link between standard storage protocols and infrastructures and the block based LoneStar RAID.

We use the FUSE userspace file system to provide a POSIX compatible file interface that can be integrated into existing storage infrastructures.

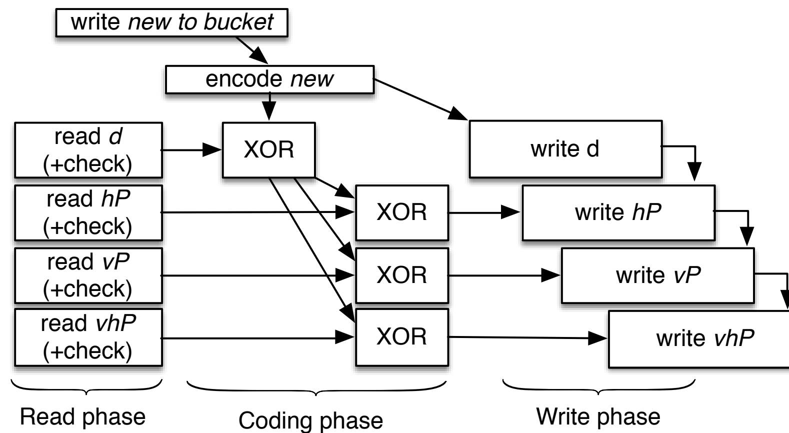


Figure 5.2: Dependencies during a RAID write. The writes are serialized to overcome the RAID-write hole problem.

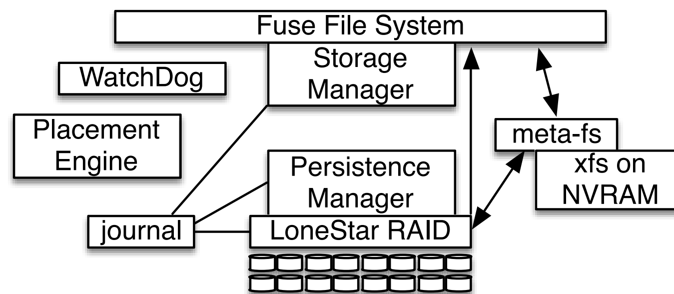


Figure 5.3: Main components of the LoneStar FS. The arrows describe the possible data flows.

The *meta-fs* component manages the file system metadata, like inodes and directory entries. Furthermore, it provides facilities for write-back caching of new files. For the *meta-fs* we use a dedicated NVRAM based partition with a common file system, like ext4 or XFS, that supports metadata-checksumming and extended attributes. We forward FUSE calls to the underlying file system and intercept calls like `open`, `read`, or `write`. New files are always written to the *meta-fs* first and are registered at the *StorageManager*. The *StorageManager* manages the fill level of the cache and the state of every file that is either new and not yet persisted to the RAID or that is already persisted but still cached.

When a new file is created, it can be written in any way to the caching *meta-fs*. However, once the last file handle for the file is closed, it cannot be updated in place anymore as the last close() triggers a complex placement decision and a set of asynchronous operations to

persist the file's extents. We do not support in-place updates of files as these operations are complex and error prone and are rarely required in archival workloads.

While new files are written, the *StorageManager* reserves a sliding window of cache space. If the file writes more data than it reserved, more cache space is acquired. If the cache is full, the *StorageManager* starts to evict the contents of already persisted files that are still cached in the *meta-fs*. Once a newly written file is closed, it is marked as immutable and the *StorageManager* assigns a unique object id (oid) to it. The oid is stored in the file's extended attributes and in a list of new objects in the database.

For journaling and a persistent state, we use a redis database that is synchronously written to the NVRAM [red]. Redis is an in-memory data structure server that provides a simple API to applications. When the database is started, the log file is replayed to recreate the last known state of the memory. It holds journaling and recovery information for the LoneStar RAID and a free/busy map of the RAID's buckets. The *StorageManager* uses the databases to store information about which files are actually present in the cache, which files need to be persisted to the RAID, and which files can be evicted from the cache based on a least recently used (LRU) strategy.

In addition to the default file stat data from the underlying *meta-fs* file system, we store a set of additional extended attributes. The stat data's file size only represents its content in the *meta-fs*. Therefore, we store the logical size of the file as `user.a.size`. Furthermore, we store the md5 hash of the file's content as `user.a.hash`. Before a new file is moved from the cache to the RAID, the *PlacementEngine* decides how many replicas shall be stored and how these replicas shall be spread to the buckets. In the simplest case, a single replica is stored on a single bucket, but more complex distributions are possible. Every file can have n replicas and every replica can consist of multiple extents that make up the files contents. The *PlacementEngine* decides how each new file should be spread to the buckets.

The *PersistenceManager* then takes these decisions, allocates the ranges on the buckets, and persists the decision to the file's extended attributes stored in the *meta-fs*. For every extent, we store a key `user.r.n-m` with value `file_offset|bucket_id|offset|length`, where n denotes the replica id and m marks the replica's extent id.

Once an extent is persisted to the RAID, we add another extended attribute `user.h.$n-$m` that contains the md5 sum of the extent's content. When a `user.h.` extent hash for every `user.r.` extent entry exists, the file is fully persisted and is marked as a cache eviction candidate in the *StorageManager*.

5.3.3 Reading files

Typically, a file access starts with a `stat` call to determine the access rights, the mode, and the size of a file followed by an `open` call and multiple consecutive `read` calls until the end of the file is reached. The LoneStar FS relies on the stat data read from the *meta-fs*'

file system but overwrites the actual file size. If the file is not cached, its stat size is 0, but the extended attribute's `user.a.size` value denotes the actual file size.

When the file's stat size equals its logical size, it can be fully read from the cache. To maintain the LRU fashion of the cache, the file's object id is updated in the eviction candidates list of the *StorageManager*.

The content of a read file can then be checked against the stored md5 hash for integrity.

If the file is not cached, or the cached copy is corrupt, the data has to be read from the LoneStar RAID. A list of the file's extents is read from its extended attributes. As long as a file is opened in FUSE, we maintain a file handle in the *StorageManager* that maintains a read and write pointer and handles to the LoneStar RAID. When data is read from the RAID, the logical read position in the file is translated to the matching extents. If there are multiple extents available that can fulfill the read, a policy decides which bucket should be read.

The LoneStar RAID is accessed through *Promises* that hide the RAIDs complexity. A read promise transparently checks and recovers data during reads and only announces an error if the data could not be verified or recovered at all. The RAID uses intra-disk redundancy to detect and recover from latent sector errors or read errors. If an extent cannot be read from a bucket, the extent or full disk is recovered from their horizontal or vertical parity groups [GMPB11].

The read data can then be checked again for integrity, using either the full file's or the extent's checksums.

5.3.4 Writing files

New files are always written as normal file system objects to the virtual *meta-fs*. During the file write to the NVRAM based file system, the file's content is hashed. If all writes occur in a serialized way, a hash is incrementally updated and the digest is created on file close. If the file was not written monotonically, the creation of the md5 hash is delayed until all file handles are closed. When the write finishes, the file is marked as immutable and registered at the *StorageManager* that associates the file with an unique oid. The file's path and oid are then pushed to the new objects database.

The *WatchDog* process runs in regular intervals and checks if there are new files that need to be persisted and if disks should be spun-up to persist cached data. First, it checks if the new objects database contains unplaced files. If so, the current fill levels and disk states of all LoneStar RAID buckets are gathered. This information and the list of unplaced files is pushed to the *PlacementEngine*, which is described in detail in Section 5.3.6. For every file, the *PlacementEngine* decides how many replicas and extents should be created and to which buckets they should be persisted and stores the decision in a placement plan that maps regions of the source file to replicas and buckets.

The *WatchDog* forwards this placement plan to the *PersistenceManager*, which executes the plan by allocating ranges on the buckets and persisting the actual placement decisions to the files extended attributes. We use allocation groups of 10 GB of size to evenly

distribute the extents over the full disk drive. Every bucket maintains a queue of outstanding work and the *PersistenceManager* enqueues links to the newly allocated extents into the according queues. When the *PersistenceManager* finishes, the *WatchDog* checks every bucket's work queue. All enqueued extents' sizes are summed up and if they hit a threshold, the extents are persisted to the RAID.

The *Cache2Disk* starts the bucket's data disk and the three parity disks hp, vp, and vhp as depicted in Figure 5.1. For every extent, a new *WritePromise* is created and its source data is read from the *meta-fs* cache and written to the RAID. When the *WritePromise* is successfully closed, the *meta-fs* file's extended attributes are updated with the md5 hash of the persisted extent. If all extents of the file are persisted, it is marked as persisted and evictable at the *StorageManager*. The *Cache2Disk* starts multiple tasks in parallel that work off the bucket's queue to overcome the penalties induced by the serialized writes as discussed in Section 5.3.1.

5.3.5 File Deletion

Metadata operations like renaming, moving, updating the access rights, or the deletion of a file do not require the actual data that is stored on the RAID but can be fully executed on the *meta-fs*. If a file is deleted, its metadata is read, all extents' ranges are marked as free and the file is deleted from the *meta-fs*. The actual deletion of the file's contents can be realized in three ways. In the simplest case, the data on the RAID is left untouched. Its range is marked as free and will, therefore, eventually be overwritten. To actually delete the content in place, two variants are possible. Either, the extents on the RAID are overwritten with random content, or the extents written to the RAID are encrypted with a key that is stored in the file's extended attributes. While the first variant can be realized by scheduling new, low priority write operations that are enqueued at the buckets' work queues, the second approach would require some additional computation and management overhead.

5.3.6 Placement Engine & Energy Management

Given the scenario of servers that host a gread number of powered off disk drives and spinning up disks on demand, the actual placement of data defines the overall energy consumption of the system. Wildani et al. [WM10] showed that the theoretical use of semantic data placement strategies can have a severe impact on the energy consumption in archival systems. Once a file is written to the RAID, the probability that its data is moved to another bucket is very low, as the costs of moving may outweigh the costs of all accesses combined. This holds especially true for archival scenarios, where the objects' lifetimes can be longer than the expected hardware lifetimes. Therefore, the first placement decision has to be as good as possible.

We integrated a *PlacementEngine* and provide as much information as possible to make a sane placement decision. In regular intervals the *WatchDog* gathers a list of all unplaced

objects and further system information. The *PlacementEngine* uses this information to map every file to one or multiple replicas and to decide to which buckets they should be stored to. The actual decision for every file can be based on the following information:

- **Filename, path, & stat data:** For every file, its full path, filename and extension as well as stat data, like the file's size, owner and group, can be used. Files owned by the same user in the same directory can be placed on the same bucket.
- **User xattr:** A strategy can integrate extended attributes on files like tags or specific ontologies.
- **Statistics:** We integrated the persisted database to provide means for statistics. The last spin-up time, the average number of disk usages or previous allocation preferences can be stored and queried. Additionally the number of files and the average extent sizes per bucket can be stored.
- **Bucket states:** The state of every disk - available, powered off, recovering - is known. Furthermore the remaining space left on a bucket is an important factor.
- **Hardware Information:** The disk drives S.M.A.R.T. data or previous counts of required recoveries as well as the actual age and spin-up counts could be taken into consideration.

New files are written to the NVRAM first and the actual placement decision is delayed. Therefore, the *PlacementEngine* is not executed in a performance critical path and can be very complex.

For the evaluation in Section 5.4, we have implemented a simple path based placement strategy. If the file's path contains a bucket's name, it is automatically placed on that bucket. This strategy eases the throughput measurements.

Further investigation of placement strategies that make use of the aforementioned available information and their simulation are subject of current research and will be discussed in future work as outlined in Section 5.5.

5.3.7 Management & RPC

To manage the system and get insights into the running file system, we provide a set of RPC calls to query state and control internals. Relevant calls include:

- **State:** Query for cache fill levels, disk states, free space on buckets. Cache state of files.
- **Cache Control**: The *WatchDog* can be triggered to enforce the placement and persisting of all outstanding files and extents or all cached files can be evicted.
- **Statistics:** Query for usage statistics like number of disk spin-ups, last access times, or throughput.

- **Parameters:** Adjust parameters like thresholds or the placement strategy.
- **Tasks and Scheduling:** Some regular tasks like disk scrubbing can be triggered or scheduled. Furthermore, complex tasks like the decommissioning of disk drives or aggregation and migration of data could be triggered.

As long as a file is not yet fully persisted to the RAID, it is registered and tracked by the *StorageManager*. Thus, a file's state can be queried and an RPC-callback can be registered. When an application finishes writing a file, a callback is registered for either the file or the containing directory. The callback is triggered when the file or the directory's content is fully persisted and it is safe to delete the original file that was copied to the archive.

5.3.8 Consistency, Reliability, and Recovery

An archival storage system possibly stores the only copy of a file. Therefore, the integrity of stored data and the reliability of the system are the most important goals to achieve. Furthermore, the portability of the data has to be considered. Even in case of hardware errors or the rare case of data loss, the vast majority of the stored data should be recoverable.

We integrate multiple paths of recovery and multiple mechanisms to check the integrity of stored data. Furthermore, we use a self-contained on disk format that can recover data in case of a metadata loss.

If an extent cannot be read correctly or a disk fails, parts or the whole disk can be recovered from the LoneStar RAID. Given its elastic fault tolerance and journaled operations, specific ranges of erroneous disks can be read and restored without waiting for whole disks to be replaced and recovered. In contrast to a typical RAID 6, where a data loss actually means the loss of all data on the RAID, the LoneStar RAID only looses data of a single non-recoverable disk drive. The MTTDL of the LoneStar RAID for various configurations was analyzed in previous work [GMPB11].

During operations like file writes or recovery, we maintain a journal that keeps track of the progress. If any error occurs, the system can be recovered to a consistent state. This can either mean to recover the RAID to a previous state or to proceed with a previously started but interrupted write task. If a write fails because of an unavailable disk drive, the *Cache2Disk* task will fail and the actual persisting is delayed.

Every extent that is written to the disk is surrounded by some extra bytes. We use an on disk format that helps to read, check and assemble extents from a disk drive without any metadata.

As shown in Figure 5.4, we use a 40 Bit magic number that helps to find and identify extents. If the disk is read from the beginning and one extent could not be read, it is very hard to find the beginning of the next one. The magic number helps to find a possible starting point. We use 128 Bit object ids. We can address 256 different replicas of a file with 2^{16} extents, each of 4 GB of size. The data part is followed by a final 32 Bit checksum over the full header and data. For every extent that is stored to the RAID, a total of 32 Byte is added.

magic	oid	replica id	extent id	data length	extent data	chksum
40	128	8	16	32	…	32

Figure 5.4: On disk format of an extent.

The LoneStar file system layer can use the extent's checksum as well as the file meta-data's checksum to verify the content. In addition to that the disk's intra-disk redundancy adds another layer of integrity checking. In combination, these different checksums can be used to verify the integrity of extents and the integrity of data that is recovered from the intra-disk redundancy or the RAID.

Disk scrubbing is a typical way to check data integrity. In regular intervals disks are spun up and all their data is read and checked for integrity. We borrow the disk scrubbing concepts presented by Storer et al. in the Pergamum system [SGMV08]. Whenever a disk is spun up, during idle times or before being shut down, the disks data is verified by its intra-disk redundancy scheme. With every spin-up, a small fraction of the disk is checked and the range is noted in the journal. Each time another range is verified and over time the full disk is checked. If for any reason the disk is not spun up for some time, a dedicated scrubbing task can be started.

Next to the RAID based integrity checking, the LoneStar stack also supports a file system based integrity checking. A single file or all files in a directory tree can be checked for integrity. All files' extents are read and their checksums are calculated. If any error is found, the extent's ranges are marked as erroneous in the RAID and scheduled for a rebuild.

5.4 Evaluation

In this section, we describe the current state of the implementation and present some performance measurements.

5.4.1 Implementation

For development and evaluation we used a 64 bit Ubuntu 12.04.4 LTS. The full LoneStar Stack is implemented in C++11. The LoneStar FS component is realized with the userspace file system FUSE. The journal and state management is implemented on a redis database. With the option `appendonly yes`, every operation that changes the database is synchronously appended to a log file. In regular intervals these log files are compressed.

The system is built around a flexible thread pool that provides asynchronous operations and long running background threads like the *WatchDog*. Wherever possible, for example during the RAID write operations depicted in Figure 5.2, operations are spread to multiple threads. To fully utilize the disk during the serialized persist operations, multiple write

tasks are started that each serialize the writes to the data and parity disks. By using up to 8 writers per bucket, the throughput is raised.

5.4.2 Evaluation Hardware

The LoneStar RAID is designed for servers that host a great number of disk drives. We used 3 JBODs of 24 Seagate Constellation 7200.2 1TB SAS drives resulting in a total of 72 disks. The JBODs were attached to a server with a 32 core Intel Xeon CPU E5-2650 - 2.00GHz with 64 GB of RAM. The redis data is persisted to an OCZ-Vertex 2 Pro.

The primary goal of the evaluation is to measure the throughput and detect the bottlenecks of the LoneStar stack. Actual server hardware that is tailored to archival workloads will be less powerful, use larger disk drives, a highly reliable NVRAM for journaling and the meta-fs, and will provide mechanisms to physically power off single disk drives.

The flexibility of the RAID schematic can find a strong configuration that balances storage efficiency, reliability and write throughput based on the discussions in previous work [GMPB11]. For the evaluation we chose a configuration that uses $x = 12$ horizontal and $y = 6$ vertical disks, which provides a raw storage utilization of 70,6%. The horizontal parity groups (hpg) use $9 + 1$ and the vertical parity groups consist of $5 + 1$ disks. The maximum write throughput of the RAID is limited by the vertical parity disks. With the chosen 12x6 configuration, a maximum of 6 buckets can be written in parallel without interfering with each other.

To simulate a fast NVRAM for the *meta-fs*, we used a RAM disk. The Linux tmpfs does not provide support for the required extended attributes. Therefore, we loop-mount an xfs formatted file that is stored in the tmpfs. This setup also allows us to individually adjust the actual cache size for each test.

5.4.3 Evaluation Design

When new data is written to the LoneStar FS, the state of the cache decides the throughput of the system. We distinguish between a) writing to an empty *meta-fs*, b) writing to an empty *meta-fs* with an ongoing persist task, c) writing to a full *meta-fs* while data is constantly persisted to the RAID and evicted from the cache, and d) data is persisted from the *meta-fs* to the RAID until no outstanding data is available. In the following, the throughput that can be measured by an application using the file system is called *fs-throughput*. Writes always hit the NVRAM, while read requests can either hit a cached file or be served from the RAID. Furthermore, we define the throughput from the NVRAM to the RAID or the RAID to the file system interface as *raid-throughput*.

When data is written to the *meta-fs*, the *PlacementEngine* decides how it is divided and spread to the buckets and the *PersistenceManager* coordinates the actual write tasks. In regular intervals the *WatchDog* checks if disks need to be started. This decision is based on a policy that defines thresholds of outstanding data per bucket or the *meta-fs* fill level.

For the evaluation, we chose a simple placement strategy that places all files in directory 'bucket_5' to the respective bucket as outlined in Section 5.3.6.

We built a tailored benchmarking environment - *lsbench* - that can make use of the placement strategy and, therefore, target specific buckets for new files. Furthermore, it uses the API described in Section 5.3.7 to gather internal state like cache fill levels and disk throughput.

5.4.4 LoneStar FS Overhead

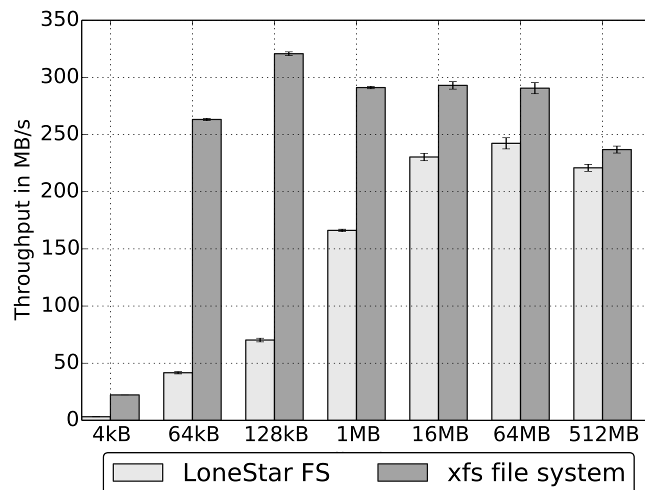


Figure 5.5: Throughput measurements to quantify the LoneStar FS overhead. Each test was executed at least 10 times and the errors mark a 95% confidence interval.

We analyzed the overhead that is added by the LoneStar FS in comparison to direct writes to a file system. We used filebench 1.4.9.1 [fil] with the pre-defined 'createfiles' personality.

The tool created 100,000 4 kB, 64 kB, and 128 kB files and 32 GB of 1 MB, 16 MB, 64 MB and 512 MB files. The target directory is either the LoneStar FS FUSE mountpoint or the xfs file system that is used as *meta-fs*. We used a dedicated xfs formatted partition on an OCZ-Vertex 2 Pro SSD. As both targets hit the same file system on the same hardware, we can quantify the overhead of the LoneStar FS. This microbenchmark implicitly measures the aforementioned fs-throughput of a). The cache was big enough to hold all written data, and the LoneStar FS was configured to not start any persist tasks. Nevertheless, the initial md5 checksum creation, extended attribute updates and placement decisions were executed.

Referencing Figure 5.5, the increasing throughput of the LoneStar FS with rising file sizes indicates a high metadata overhead. In addition to the typical file system operations

for newly written files like inode creation and space allocation, the LoneStar FS adds additional overhead. On file creation, the *StorageManager* ensures the availability of free cache space. During writes, the md5 sum of the file is incrementally calculated. Finally, when the file is created, it is registered at the journal and its extended attributes are updated, which results in additional synchronous IO operations. Ultimately, the FUSE overhead, as analyzed by Rajgarhia and Gehani[RG10], becomes visible. Nevertheless, with rising file sizes the relative performance loss towards a local file system becomes smaller.

5.4.5 Read Performance

To measure the actual read performance of the system, we created 8 GB of files of fixed sizes on each bucket. The used extent alloction scheme evenly distributes the files over the full disk drive. The the LoneStar FS was flushed and all cached data was evicted so that every read from an application hit the disk. Before the actual read test, the operating system page cache also got flushed.

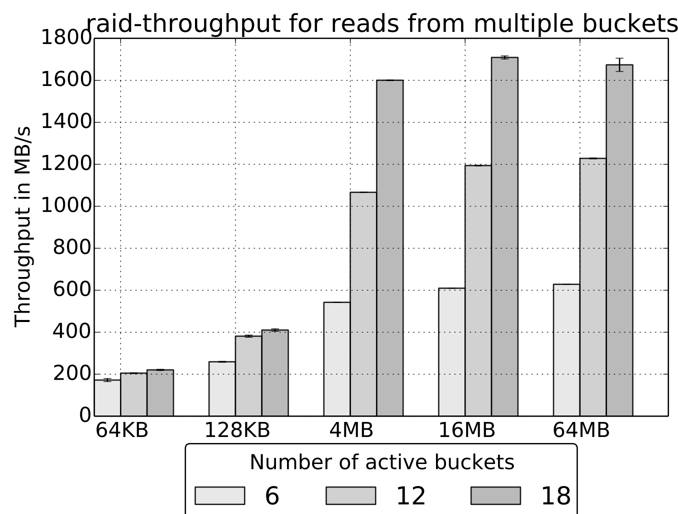


Figure 5.6: Throughput of file reads from 6, 12, 18 buckets in parallel. Data is read from disks and checked for integrity. Each test was executed at least 10 times and the errors mark a 95% confidence interval.

The test started one reader process per bucket that read all files from the bucket. During the read, the LoneStar RAID checks every block for integrity using its IDR code. The results show that the system is capable of reading disks close to their maximum capabilities.

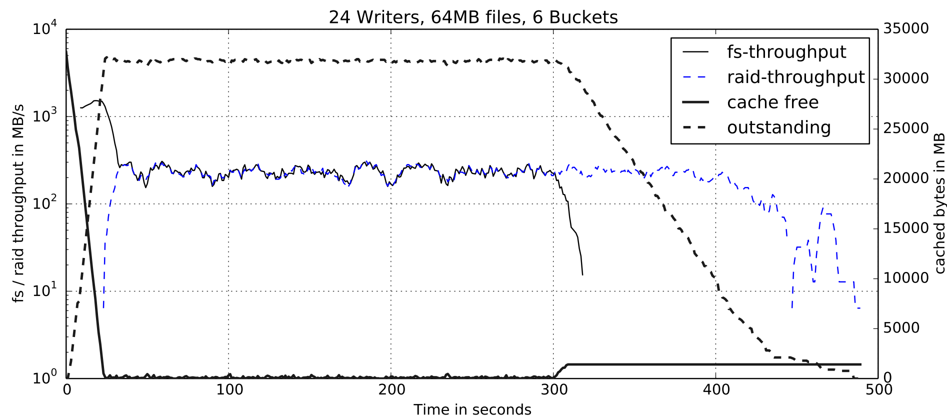


Figure 5.7: 24 clients write 64 MB files evenly distributed to 6 buckets to a clean cache. The thin lines of the left axis depict a 10 second rolling mean of the fs / and raid-throughputs, while the thick lines present cache internals. The benchmark stops when all data is fully persisted to the disks.

5.4.6 Write Performance

Figure 5.7 depicts the fs- and raid-throughput during a single, complete write benchmark run. The test starts with about 1.4 GB/s in phase a) until the cache is filled and files need to be persisted and evicted from the cache. Whenever a cache eviction is triggered, at least 128 MB of cache are freed, which also explains the raised free cache at 300 seconds. The time period from when the cache is full and the *PersistenceManager* starts the disks till the last file write finishes on the file system characterizes the steady state of the system c). After that, the disks persist all outstanding data.

Next, we analyze the raid-throughput during the previously described cases c) and d). The tests ran with 4 processes per bucket that created files of fixed sizes for 300 seconds with a meta-fs size of 40 GB. As soon as a bucket has 4 GB of outstanding data, it is started to persist to disk. The measurement of the fs-throughput starts when the first disk drive starts to persist data and runs until the cache is fully flushed to the disk.

The results in Figure 5.8 shows the performance penalty induced by the LoneStar RAID. Due to its interwoven parity groups and serialized writes that facilitate the strong consistency guarantees, the write performance is limited to half of the disks' speeds and to the throughput of the shared parity disks. The tests were designed to write to 6 buckets that do not share any disk. Accordingly, the tests with 8 buckets share vertical parity disks.

The 64 MB and 128 MB writes show the best write throuhput. By writing the two additional buckets, the overall performance is getting worse. For the smaller writes, on the other hand, the performance is rising for 8 buckets compared to the 6 buckets result. The system is working highly parallel and multiple threads share the access to the disk drives. To optimize the throughput, a read-encode-write pipeline is required that per-

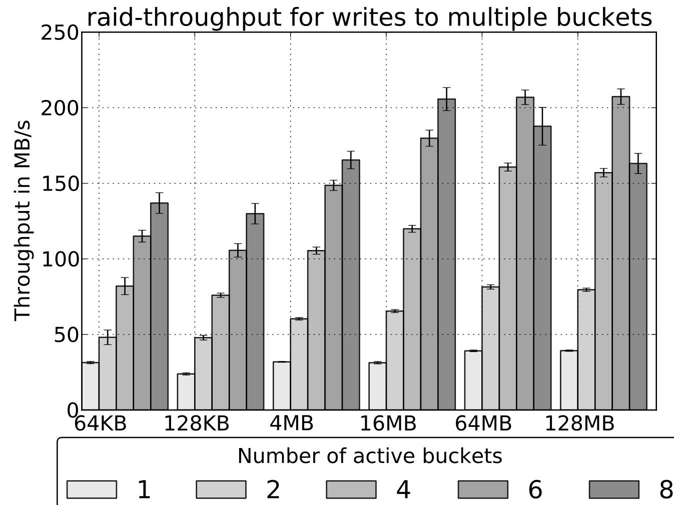


Figure 5.8: Raid-throughput during steady state of writing to 1, 2, 4, 6, 8 buckets in parallel with different fixed file sizes. Each test was executed at least 10 times and the errors mark a 95% confidence interval.

fectly matches the dependencies depicted in Figure 5.2 and fully utilizes all 4 involved disks through the persist process. During the benchmarks we observed that the CPU does not become a bottleneck. For every incoming file an md5 sum is created, for every written extent an md5 sum is created and every new block that is written to the RAID is IDR encoded and woven into the two-dimensional RAID. The file system in sum uses at most 800% of the 32 available cores for parallel writes to 8 buckets. The results show, that there is some space left for improvement. Given the disk capabilities and remaining options for optimization we are optimistic to see a write throughput close to 300 MB/s for the given 72 disk configuration.

5.5 Conclusion & Future Work

In this paper we presented the architecture of the full LoneStar stack consisting of a new file system that is tailored to use the LoneStar RAID. The POSIX compatible interface uses a write-back cache and asynchronous operations to aggregate update operations and move data to target disks chosen by semantic data placement strategies. With this architecture we overcame the challenges imposed by the LoneStar RAID architecture and created a storage building block that can be used in various scenarios. The performance evaluation revealed the need for improvements for small files but also showed that a high throughput close to the hardware limits is possible.

The next step is the further investigation of data placement strategies to evaluate their actual impact on the overall energy consumption of the system. We will use a trace based simulation environment that builds on the results of this paper and available disk energy models to evaluate and measure different placement strategies and hardware configurations.

Today, the storage requirements of digital archives exceed the capacity that can be offered by a single server instance. Therefore, we further investigate how multiple LoneStar Stacks can be combined to build a scaling archive environment.

As an additional step, we will investigate the ramifications of replacing md5 with the BLAKE2 hashing system, which is optimized for parallel high data througput [ANWOW13]. It also provides tree based hashing, which supports in place updates and, therefore, helps to maintain checksums for large objects or even full disk content.

6 | Conclusion

In this thesis we investigated if and how a disk-based storage system can be built that is able to compete or even outperform a tape-based solution for archival workloads. In the following, we first summarize the undertaken steps and give an answer to this question based on the findings of the previous chapters. Finally, we consider additional avenues of research.

In Chapter 2, we analyzed the storage landscape of the European Centre for Medium-Range Weather Forecasts (ECMWF), which showed the shortcomings of tape-based storage in large-scale archival scenarios in terms of performance and cost-efficiency. The investigated data center mainly hosts two storage systems. ECFS is a general purpose archive for user data and MARS is an object database for meteorological data. Both systems primarily store their data on tape, but use large disk-based caches to deliver the required access performance. Though achieving cache hit ratios above 80%, the tape drives are constantly working close to saturation. Most of the data accesses run in batched processes, so that the access latency of tape is no immediate bottleneck for data processing. Nevertheless, many parallel requests or requests that require many different data items can result in waiting times of many minutes or even hours. A more responsive solution would enable even more complex models and ad-hoc queries over the full archived content. While tape is considered to be cheap mass storage, the use case of the ECMWF needs to reconsider this statement. To deliver the required data rates, many expensive tape drives are required, as well as the disk-based caches. Considering these additions to the storage stack, a tailored solution based on-disks might compete on a price-level while outperforming the tape-based architecture.

In Chapter 3, we investigated the reliability and energy models of a single disk drive. We evaluated the impact of intra-disk redundancy codes (IDR) on both performance and energy consumption of the resulting storage stack. The result was a better understanding of the implications of an applied IDR and the scrambling mechanism to improve cheap codes [GSB+11]. This work led to the applied 16+1 SPC code in the LoneStar storage system as it provides a good trade-off between error detection capabilities, recoverability, processing throughput, and storage overhead.

Chapter 4 described the two-dimensional LoneStar RAID scheme. It incorporates IDR to improve the reliability of each single disk drive of the array and was designed for systems that host many disk drives and workloads where data is mostly written once and

read sometimes. Writes are complex as every data disk is interwoven into multiple parity groups and the on-disk's parities have to be updated. Reading data off a disk is cheap and secure as only a single disk needs to be spun up and the IDR scheme provides an in-flight integrity check. To overcome the shortcomings of the RAID scheme, we proposed a journaling approach that tracks every operation on the RAID and the state of errors and recovery operations. Furthermore, we developed a cache strategy that collects and merges parity writes. This mechanism can reduce the amount of data that needs to be written to the parity disks and can collect and delay parity updates until the cache is full. Aggregating writes and delaying updates to parity disks can save many spin-ups, while preserving the reliability of the RAID.

Chapter 5 investigated the feasibility of a file system on top of a LoneStar RAID. In contrast to the block-based RAID evaluation of Chapter 4, the resulting stack moves the caching into the file system layer and reorganizes the writes to the RAID's disks. We present a fully working POSIX compatible file system that integrates a hierarchical storage management approach. New files are always written to the cache first and aggregated for batched writes to the RAID. A background process oversees all not-yet persisted data and decides where and when files should be moved from the cache to the LoneStar RAID. The system incorporates semantic data placement strategies to decide how the files are spread over the available data disks.

In conclusion, this thesis developed and described a fully working disk-based building block that can be extended and used to build large-scale digital archives that host both, active as well as cold data. The system incorporates important features like intra-disk redundancy, a self-describing on-disk format, and object and extent checksums that provide end-to-end data integrity. We also integrate disk scrubbing to check the disks in regular intervals to detect erroneous storage media. If any errors are found that cannot be compensated with the IDR, the RAID scheme can be used to recover data. The elastic fault tolerance and the applied operation journaling provide an always consistent system state, even in the case of multiple parallel disk errors. Data is not striped over multiple disks, which provides implicit fault isolation. If a data disk cannot be recovered, only the disk's data is lost, while the remaining array is still available.

The write performance of the system is mainly bounded by the throughput and size of the used NVRAM. If the cache is full, all parallel processes that move data from the cache to the RAID define the resulting throughput. By default, related blocks are stored on the same data disk but they can be spread to multiple data disks on demand. Consequently, the number of involved disks bound the read performance.

The system integrates important mechanisms to successfully realize a MAID approach that enables extreme energy savings, as unneeded components can be powered down when being idle. First, the RAID is designed to require only a single disk to read a file. Also, semantic data placement strategies are incorporated into the system, which help to aggregate and place related data. In the best case, all accessed files of a user session can be served

from a single disk drive. Furthermore, the caching scheme aggregates RAID updates for batch processing to reduce the required and expensive disk spin-ups.

The system is also efficient from an economical standpoint. It is very dense in terms of possible disk drives per server and provides a considerable storage overhead. Configurations of the RAID scheme can be found that use a high number of disks, provide a high reliability, and allow a high parallelism with a storage overhead of 136%. The system is energy efficient as all idle disks can be spun down, which is especially important for cold storage scenarios. But also for active archive environments, the proposed LoneStar system can be a good fit. On demand, it can deliver high read performance, but it can also save energy by shutting off idle disks. In contrast to the worst case access latencies for tape that can stack up infinitely, the expected access latencies for LoneStar are much less. In the best case, the data can be served from the cache or an already spinning disk. A disk spin-up typically takes 10 seconds, which renders the average access time. In the worst case, multiple disk drives fail so that multiple other disks need to be spun up to start a recovery.

We conclude that this thesis justifies and describes the architecture of a novel, highly reliable, and energy-efficient disk-based storage system that can be used in cold storage as well as active archive environments.

6.1 Future Work

This work describes a fully working archival system. Nevertheless, there are a number of planned augmentations and opportunities for future research.

Though the LoneStar Stack is designed to outperform tape-based architectures on performance and economical scale, an analysis of the latter is outstanding. The LoneStar system incorporates multiple mechanisms to support semantic data placement strategies. For any workload, this placement strategy will define the resulting performance and energy consumption of the LoneStar system. Both, the economical analysis as well as an in depth evaluation of the placement strategies would strongly benefit from more traces of real usage scenarios. Only a few traces and descriptions of workloads exist for today's archival systems. The ECMWF study of this thesis adds a trace to the public, but more systems need to be analyzed to foster research on the efficiency of active and cold archival stores. To make use of the traces, a storage system simulator would be required that models flash, disk, and tape systems and provides in-depth energy models as well as performance statistics. There is no single simulation environment that covers all the relevant areas up to now. But previous work could be combined and improved [BSSG08, YLZ13]. Given such a system, a LoneStar archive with a particular placement strategy and a tape-based architecture could be simulated and compared.

The current LoneStar system describes a single server with many disks. To scale to today's storage requirements, many servers are required. This opens the question how to

distribute data and accesses over many LoneStar hosts. Alternatively, the system can be integrated into existing distributed file systems as an archival tier.

Many additional features for the storage system were discussed and would improve the described storage stack. First, the integration of a tree hash like BLAKE2 [ANWOW13] would improve the integrity of the system. The currently applied MD5, or SHA1 are Merkle–Damgård hash functions. These hash schemes can only be calculated for a serial stream of bits of arbitrary length. Given a 1 GB file for example, its hash can be calculated efficiently while the file is read or written sequentially. If a single bit is updated, the full file needs to be processed again to recalculate the hash. By applying a tree hash like BLAKE2, the file can be segmented into multiple blocks that can be hashed and processed in parallel. The sum of the segments' hashes is the hash of the full file. Therefore, an update only requires the reprocessing of the updated segment. Compared to MD5, BLAKE2 is more complex to use and to integrate, but offers a better performance, the possibility to parallelize calculations, and the aforementioned flexibility of tree-based hashing.

Another feature that can reduce the number of physically stored bytes is the integration of a compression layer. Algorithms like LZ4[1] or snappy[2] can be seamlessly integrated into the file system layer.

Another reduction of the stored data can be achieved with deduplication. Blocks of newly stored files are hashed and compared to all previously stored blocks. If a block already exists, it is not stored again. Therefore, all duplicate data can be eliminated. Typical deduplication systems work on all stored data of a system. This is bad for the LoneStar architecture as it tries to place related data on the same disks. With applied deduplication, blocks of files can be spread over all data disks. The locality of data blocks that make up a file is even more important when deduplication should be applied to tape-based systems. This topic was investigated by Gharaibeh et al. who proposed the DedupT deduplication system for tape [GCL+14]. The assumption is that with growing storage media, also the deduplication ratio within each medium is growing. With today's disk drives of 6 TB and even bigger drives on the horizon, the scenario of deduplication on MAID storage might become relevant.

Last but not least, the system can seamlessly integrate an encryption scheme. Such a scheme can either be integrated into an application that stores data to the LoneStar system, or it can be integrated into the file system layer. As a result, all data stored on disk would be encrypted.

The final dedup-hash-encrypt pipeline that could be integrated into the LoneStar file system could work as follows. First, new files are hashed for a deduplication application. Then they are compressed and encrypted while being written to the cache. Finally, the cached data is hashed again to provide the file integrity checksums of the data on the RAID.

[1] https://code.google.com/p/lz4/
[2] https://code.google.com/p/snappy/

This list of future work only included augmentations from a software perspective. It is also important to match the feasibility of the proposed storage stack to upcoming technology advancements. While cost per gigabyte for disk drives dropped exponentially over the last three decades, this trend stopped around 2010. Gupta et al. [GWR+14] analyzed the economic perspective of disk vs. flash media for archival storage and predict that flash will become increasingly important for archival scenarios within the next years. LoneStar was tailored for disk-based storage media, exploits their characteristics and integrates means to overcome typical failure scenarios. Nevertheless, it can be adapted to use flash instead of disks. Flash does not need to spin-up and is immediately accessible. Therefore, the system would be more responsive and the size of the expensive caching layer could be decreased, as writes would be directly forwarded to the target storage media. Like disks, flash-based drives will also either fail at all or suffer from IO errors like failing reads or corrupt data. Therefore, the IDR and the LoneStar RAID scheme will help to improve the overall reliability with a high storage efficiency. Nevertheless, the failure characteristics of flash-media need to be investigated to fully adapt the reliability mechanisms.

In the contstant advancement of disk drives, the shingled magnetic recording technology delineates as the next promising large step towards higher disk capacities. This technology is visible in the research community for some years now and first disks are now sold by major disk manufacturers [WWKM09, ALM+10, CSG+10, GG11, PRH+12, WZZ+12, JXC+14]. In general, shingled write disks (SWD) increase the density of disk platters by shingling adjacent tracks. Disks are now organized into bands that each contain multiple overlapping tracks, with a small safety gap between adjacent bands. Data can still be randomly read, but a write to a track possibly also alters other tracks within the same band. This new constraint fundamentally changes the mechanisms for writes. In general, two abstract modes have been identified to be feasible. On the one hand, data can be appended within a band without affecting previously written tracks. Strictly following this constraint results in log-structured systems and the requirement for garbage collection schemes, but offers a good performance. Alternatively, in-place updates to SWDs are possible. But they require a read-modify-write cycle, as writes within a band overlap and partially overwrite surrounding tracks. The consequence is to read all tracks of a band, manipulate the data in memory and write the band back to disk. Previous work showed that, if the disk geometry is exactly known, the overall throughput can be improved, as data objects can be aligned to the disks' native sector and track sizes [SGLG02, QMW08]. For SWDs, the knowledge of track sizes and band boundaries does not only help to improve the performance, but is indispensable to not lose data. The logic how to write to the shingled disks can either be abstracted and hidden within the disk itself, within an operating system layer, or it could be fully exposed to an application like LoneStar.

We think that the LoneStar RAID can be well adapted to SWDs. The main challenge is to fit the iPGGroup concept to the shingled disk's bands. The RAID scheme requires a RMW-cycle to update data in place. Now, the full surrounding band is read, updated and written back. In the best case, the IDR scheme and the resulting iPG size is fully adapted

to the band size. Depending on the number of tracks and the resulting band sizes, the scrambling approach to improve the IDR needs to be reinvestigated. Scrambling can be applied, as long as all iPGs of the group reside within the same band. If they span multiple bands, RMW-cycles on all bands are required which dramatically decreases the disk drive's performance.

In conclusion, we think that the proposed LoneStar architecture works well with the storage media that is available today. Furthermore, the discussion around flash-based archives and shingled write disks showed that this work can also be adapted to upcoming technology advancements.

Glossary

DFT Disk failure tolerancy

CAGR Compound annual growth rate

ECFS ECMWF file system

ECMWF European Centre for Medium-Range Weather Forecasts

FDB Field database

FUSE Filesystem in userspace

HPG Horizontal parity group

HPSS High performance storage system

IDR Intra-disk redundancy

iPG Intra-disk parity group

LSE Latent sector error

MAID Massive Array of Idle Disks

MARS Meteorological Archival and Retrieval System (MARS)

MTTDL Mean time to dataloss

NVRAM Non-volatile random-access memory

POSIX Portable operating system interface

RAID Redundant array of independent disks

RMW-cycle Read-modify-write cycle

SPC Single Parity Check

SWD Shingled write disk

VPG Vertical parity group

Bibliography

[ABDL07] N. Agrawal, W. J. Bolosky, J. R. Douceur, and J. R Lorch. A Five-year Study of File-system Metadata. *ACM Transactions on Storage (TOS)*, 3(3), 2007.

[ALM+10] A. Amer, D. D. E. Long, E. L. Miller, J.-F. Pâris, and T. S. J. Schwarz. Design issues for a shingled write disk system. In *Proc. of the 26th IEEE Conference on Mass Storage Systems and Technologies (MSST)*, 2010.

[AMS10] I. F. Adams, E. L. Miller, and M. W. Storer. Examining Energy Use in Heterogeneous Archival Storage Systems. In *Proc. of the 18th IEEE International Symposium on Modeling, Analysis, and Simulation (MASCOTS)*, 2010.

[ANWOW13] J.-P. Aumasson, S. Neves, Z. Wilcox-O'Hearn, and C. Winnerlein. BLAKE2: Simpler, Smaller, Fast as MD5. In *Applied Cryptography and Network Security*, volume 7954. 2013.

[ASM12] I. F. Adams, M. W. Storer, and E. L. Miller. Analysis of Workload Behavior in Scientific and Historical Long-Term Data Repositories. *ACM Transactions on Storage (TOS)*, 8(2), 2012.

[AvMT12] R. Appuswamy, D. C. van Moolenbroek, and A. S. Tanenbaum. Integrating flash-based SSDs into the storage stack. In *Proc. of the 28th IEEE Conference on Mass Storage Systems and Technologies (MSST)*, 2012.

[AXF+12] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload Analysis of a Large-scale Key-value Store. In *Proc. of the 12th ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2012.

[B́66] L.A. Bélády. A Study of Replacement Algorithms for a Virtual-storage Computer. *IBM Systems Journal*, 5(2), 1966.

[BADAD+08] L. N. Bairavasundaram, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, G. R. Goodson, and B. Schroeder. An analysis of data corruption in the storage stack. *ACM Transactions on Storage (TOS)*, 4(3), 2008.

[BBBM95] M. Blaum, J. Brady, J. Bruck, and J. Menon. EVENODD: An Efficient Scheme for Tolerating Double Disk Failures in RAID Architectures. *IEEE Transactions on Computers*, 44(2), 1995.

[BBD+14] S. Balakrishnan, R. Black, A. Donnelly, P. England, A. Glass, D. Harper, S. Legtchenko, A. Ogus, E. Peterson, and A. Rowstron. Pelican: A building block for exascale cold data storage. In *Proc. of the 11th Conference on Operating Systems Design and Implementation (OSDI)*, 2014.

[BBL06] T. Bisson, S. A. Brandt, and D. D. E. Long. NVCache: Increasing the effectiveness of disk spin-down algorithms with caching. In *Proc. of the 14th IEEE International Symposium on Modeling, Analysis, and Simulation (MASCOTS)*, 2006.

[BCF+99] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web Caching and Zipf-like Distributions: Evidence and Implications. In *Proc. of the 18th IEEE International Conference on Computer and Communications (INFOCOM)*, 1999.

[BGPS07] L. N. Bairavasundaram, G. R. Goodson, S. Pasupathy, and J. Schindler. An analysis of latent sector errors in disk drives. In *Proc. of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2007.

[BKM05] M. Baker, K. Keeton, and S. Martin. Why Traditional Storage Systems Don't Help Us Save Stuff Forever. In *Proc. of the 1st Workshop on Hot Topics in System Dependability (HotDep)*, 2005.

[BSR+06] M. Baker, M. Shah, D.S.H. Rosenthal, M. Roussopoulos, P. Maniatis, T.J. Giuli, and P. Bungale. A Fresh Look at the Reliability of Long-term Digital Storage. In *Proc. of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys)*, 2006.

[BSS00] A. Brinkmann, K. Salzwedel, and C. Scheideler. Efficient, distributed data placement strategies for storage area networks. In *Proc. of the 12th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, 2000.

[BSSG08] J. S. Bucy, J. Schindler, S. W. Schlosser, and G. R. Ganger. The disksim simulation environment version 4.0 reference manual (cmu-pdl-08-101). *Parallel Data Laboratory*, 2008.

[CEG+04] P. F. Corbett, R. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar. Row-Diagonal Parity for Double Disk Failure Correction. In *Proc. of the 3rd USENIX Conference on File and Storage Technologies (FAST)*, 2004.

[CG02] D. Colarelli and D. Grunwald. Massive Arrays of Idle Disks for Storage Archives. In *Proc. of the ACM/IEEE conference on Supercomputing (SC)*, 2002.

[CGK+10] D. Chen, G. Goldberg, R. Kahn, R. Kat, and K. Meth. Leveraging Disk Drive Acoustic Modes for Power Management. In *Proc. of the 26th IEEE Conference on Mass Storage Systems and Technologies (MSST)*, 2010.

[CGN02] D. Colarelli, D. Grunwald, and M. Neufeld. The Case for Massive Arrays of Idle Disks (MAID). In *Proc. of the USENIX Conference on File and Storage Technologies (FAST)*, 2002.

[CPB03] E. V. Carrera, E. Pinheiro, and R. Bianchini. Conserving disk energy in network servers. In *Proc. of the 17th ACM/IEEE conference on Supercomputing (SC)*, 2003.

[CSG+10] Y. Cassuto, M. A. A. Sanvido, C. Guyot, D. R. Hall, and Z. Z. Bandic. Indirection systems for shingled-recording disk drives. In *Proc. of the 26th IEEE Conference on Mass Storage Systems and Technologies (MSST)*, 2010.

[CSGK11] Y. Chen, K. Srinivasan, G. R. Goodson, and R. H. Katz. Design Implications for Enterprise Storage Systems via Multi-Dimensional Trace Analysis. In *Proc. of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[CWMX10] M. Chrobak, G. J. Woeginger, K. Makino, and H. Xu. Caching Is Hard – Even in the Fault Model. In M. de Berg and U. Meyer, editors, *Algorithms – ESA 2010*, volume 6346 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2010.

[DB99] J. R. Douceur and W. J. Bolosky. A large-scale study of file-system contents. In *Proc. of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 1999.

[DEH+08] A. Dholakia, E. Eleftheriou, X.-Y. Hu, I. Iliadis, J. Menon, and K. K. Rao. A New Intra-Disk Redundancy Scheme for High-Reliability RAID Storage Systems in the Presence of Unrecoverable Errors. *ACM Transactions on Storage (TOS)*, 4(1), 2008.

[DKM94] F. Douglis, P. Krishnan, and B. Marsh. Thwarting the power-hungry disk. In *Proc. of the USENIX Winter Technical Conference*, 1994.

[ECM14] ECMWF. ECMWF Data Handling System. `www.ecmwf.int/en/computing/our-facilities/data-handling-system`, September 2014.

[EK02] K. M. Evans and G. H. Kuenning. A Study of Irregularities in File-Size Distributions. In *Proc. of the International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS)*, 2002.

[fbf] flashcache. `github.com/facebook/flashcache`.

[FHC11] W. Felter, A. Hylick, and J. Carter. Reliability-Aware Energy Management for Hybrid Storage System. In *Proc. of the 27th IEEE Conference on Mass Storage Systems and Technologies (MSST)*, 2011.

[fil] filebench. `filebench.sourceforge.net`.

[FMR12] J. C. Frank, E. L. Miller, and I. F. Adams D. C. Rosenthal. Evolutionary trends in a supercomputing tertiary storage environment. In *Proc. of the 20th IEEE International Symposium on Modeling, Analysis, and Simulation (MASCOTS)*, 2012.

[GAKF03] S. Gurumurthi, S. Anand, M. Kandemir, and H. Franke. DRPM: dynamic speed control for power management in server class disks. In *Proc. of the 30th Annual International Symposium on Computer Architecture*, 2003.

[GALM07] P. Gill, M. F. Arlitt, Z. Li, and A. Mahanti. Youtube Traffic Characterization: A View from the Edge. In *Proc. of the 7th ACM SIGCOMM Conference on Internet Measurement (IMC)*, 2007.

[GBSB14] M. Grawinkel, G. Best, M. Splietker, and A. Brinkmann. LoneStar Stack: Architecture of a Disk-Based Archival System. In *Proc. of the 9th IEEE International Conference on Networking, Architecture, and Storage (NAS)*, 2014. ©2014 IEEE. Reprinted, with permission, from M. Grawinkel and G. Best and M. Splietker and A. Brinkmann, LoneStar Stack: Architecture of a Disk-Based Archival System, August 2014.

[GCL+14] A. Gharaibeh, C. Constantinescu, M. Lu, A. Sharma, R. R. Routray, P. Sarkar, D. Pease, and M. Ripeanu. DedupT: Deduplication for Tape Systems. 2014.

[GG11] G. Gibson and G. Ganger. Principles of operation for shingled disk devices. In *Proc. of the 3nd USENIX Conference on Hot Topics in Storage and File Systems (HotStorage)*, 2011.

[GHK+89] G. Gibson, L. Hellerstein, R. Karp, R. Katz, and D. Patterson. Coding techniques for handling failures in large disk arrays. In *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 1989.

[GLM+08] K. M. Greenan, D. D. E. Long, E. L. Miller, T. Schwarz, and J. J. Wylie. A Spin-Up Saved is Energy Earned: Achieving Power-Efficient, Erasure-Coded Storage. In *Proc. of the 4th Workshop on Hot Topics in System Dependability (HotDep)*, 2008.

[GLW10] K. M. Greenan, X. Li, and J. J. Wylie. Flat XOR-based erasure codes in storage systems: Constructions, efficient recovery, and tradeoffs. In *Proc. of the 26th IEEE Conference on Mass Storage Systems and Technologies (MSST)*, 2010.

[GMB10] Y. Gao, D. Meister, and A. Brinkmann. Reliability Analysis of Declustered-Parity RAID 6 with Disk Scrubbing and Considering Irrecoverable Read Errors. In *Proc. of the 5th IEEE International Conference on Networking, Architecture, and Storage (NAS)*, 2010.

[GMPB11] M. Grawinkel, H. Dömer M. Pargmann, and A. Brinkmann. LoneStar: An Energy-Aware Disk Based Long-Term Archival Storage System. In *Proc. of the 17th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, 2011.

[GNB15] M. Grawinkel, L. Nagel, and A. Brinkmann. LoneStar RAID: Massive Array of Offline Disks for Archival Systems. *Submitted to ACM Transactions on Storage (TOS)*, 2015.

[GNM+15] M. Grawinkel, L. Nagel, M. Mäsker, F. Padua, A. Brinkmann, and L. Sorth. Analysis of the ECMWF Storage Landscape. *Proc. of the 13th USENIX Conference on File and Storage Technologies (FAST)*, 2015.

[GPG+11] J. Guerra, H. Pucha, J. S. Glider, W. Belluomini, and R. Rangaswami. Cost effective storage using extent based dynamic tiering. In *Proc. of the USENIX Conference on File and Storage Technologies (FAST)*, 2011.

[Gre94] P. M. Greenawalt. Modeling power management for hard disks. In *Proc. of the IEEE International Symposium on Modeling, Analysis, and Simulation (MASCOTS)*, 1994.

[GSB+11] M. Grawinkel, T. Schäfer, A. Brinkmann, J. Hagemeyer, and M. Porrmann. Evaluation of Applied Intra-Disk Redundancy Schemes to Improve Single Disk Reliability. In *Proc. of the 19th IEEE International Symposium on Modeling, Analysis, and Simulation (MASCOTS)*, 2011. ©2011 IEEE. Reprinted, with permission, from M. Grawinkel and T. Schäfer and A. Brinkmann and J. Hagemeyer and M. Porrmann, Evaluation of Applied Intra-Disk Redundancy Schemes to Improve Single Disk Reliability, July 2011.

[GWC+08] J. Gim, Y. Won, J. Chang, J. Shim, and Y. Park. DIG: Rapid Characterization of Modern Hard Disk Drive and Its Performance Implication. In *Proc. of the 5th Internernational Workshop on Storage Network Architecture and Parallel I/Os (SNAPI)*, 2008.

[GWR+14] P. Gupta, A. Wildani, D. Rosenthal, E. L. Miller, I. F. Adams, C. Strong, and A. Hospodor. An Economic Perspective of Disk vs. Flash Media in Archival Storage. In *Proc. of the 22th IEEE International Symposium on Modeling, Analysis, and Simulation (MASCOTS)*, 2014.

[Haf05] J. L. Hafner. WEAVER Codes: Highly Fault Tolerant Erasure Codes for Storage Systems. In *Proc. of the 4th USENIX Conference on File and Storage Technologies (FAST)*, 2005.

[HBvR+13] Q. Huang, K. Birman, R. van Renesse, W. Lloyd, S. Kumar, and H. C. Li. An Analysis of Facebook Photo Caching. In *Proc. of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[HG92] M. Holland and G. A. Gibson. *Parity declustering for continuous operation in redundant disk arrays*, volume 27. ACM, 1992.

[HLM94] D. Hitz, J. Lau, and M. Malcolm. File system design for an NFS file server appliance. In *Proc. of the USENIX Winter Technical Conference*, 1994.

[HLS96] D. P. Helmbold, D. D. E. Long, and B. Sherrod. A dynamic disk spin-down technique for mobile computing. In *Proc. of the 2nd Annual International Conference on Mobile Computing and Networking (MobiCom)*, 1996.

[HR06] J. L. Hafner and K. K. Rao. Notes on reliability models for non-MDS erasure codes. Technical report, IBM Research Division, 2006.

[HS09] A. Hylick and R. Sohan. A methodology for generating disk drive energy models using performance data. *Energy (Joules)*, 80, 2009.

[HSRJ08] A. Hylick, R. Sohan, A. Rice, and B. Jones. An analysis of hard drive energy consumption. In *Proc. of the IEEE International Symposium on Modeling, Analysis, and Simulation (MASCOTS)*, 2008.

[HSX+12a] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin. Erasure Coding in Windows Azure Storage. In *Proc. of the Annual Technical Conference (ATC)*, 2012.

[HSX+12b] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, S. Yekhanin, et al. Erasure coding in windows azure storage. In *Proc. of the Annual Technical Conference (ATC)*, 2012.

[IHHE08] I. Iliadis, R. Haas, X.-Y. Hu, and E. Eleftheriou. Disk Scrubbing versus Intra-Disk Redundancy for High-Reliability Raid Storage Systems. In *Proc. of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2008.

[JK09] E. Jaffe and S. Kirkpatrick. Architecture of the Internet Archive. In *Proc. of the ACM Israeli Experimental Systems Conference (SYSTOR)*, 2009.

[JRU+10] L. Jones, M. Reid, M. Unangst, G. Gibson, and B. Welch. Panasas Tiered Parity Architecture. White Paper, 2010.

[JXC+14] C. Jin, W.-Y. Xi, Z.-Y. Ching, F. Huo, and C.-T. Lim. Hismrfs: A high performance file system for shingled storage array. In *Proc. of the 30th IEEE Conference on Mass Storage Systems and Technologies (MSST)*, 2014.

[KMHB12] J. Kaiser, D. Meister, T. Hartung, and A. Brinkmann. Esb: Ext2 split block device. In *Proc. of the 18th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, 2012.

[KV08] I. Koltsidas and S. D. Viglas. Flashing up the storage layer. In *Proc. of the 34th International conference on Very Large Data Bases (VLDB Endowment)*, 2008.

[Lev09] A. Leventhal. Triple-Parity RAID and Beyond. *ACM Queue*, 7(11), 2009.

[LKHA94] K. Li, R. Kumpf, P. Horton, and T. E. Anderson. A quantitative analysis of disk drive power management in portable computers. In *Proc. of the USENIX Winter Technical Conference*, 1994.

[LMB10] P. Lensing, D. Meister, and A. Brinkmann. hashfs: Applying Hashing to Optimize File Systems for Small File Reads. In *Proc. of the 6th Internernational Workshop on Storage Network Architecture and Parallel I/Os (SNAPI)*, 2010.

[Los14] Los Alamos National Laboratory. Archive Data to Support and Enable Computer Science Research. `institutes.lanl.gov/data/archive-data`, 2014.

[LPGM08] A. W. Leung, S. Pasupathy, G. Goodson, and E. L. Miller. Measurement and Analysis of Large-scale Network File System Workloads. In *Proc. of the Annual Technical Conference (ATC)*, 2008.

[LW04] D. Li and J. Wang. EERAID: Energy Efficient Redundant and Inexpensive Disk Array. In *Proc. of the 11th ACM SIGOPS European Workshop (SIGOPS)*, 2004.

[MAFM12] B. Madden, I. F. Adams, J. Frank, and E. L. Miller. Analyzing User Behavior: A Trace Analysis of the NCAR Archival Storage System. Technical Report UCSC-SSRC-ssrctr-12-02, University of California, Santa Cruz, 2012.

[MEK+11] A. Miranda, S. Effert, Y. Kang, E. L. Miller, A. Brinkmann, and T. Cortes. Reliable and Randomized Data Distribution Strategies for Large Scale Storage Systems. In *Proc. of the 18th High Performance Computing Conference (HiPC)*, 2011.

[MJLF84] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for unix. *ACM Transactions on Computer Systems (TOCS)*, 2(3), 1984.

[MKB+12] D. Meister, J. Kaiser, A. Brinkmann, T. Cortes, M. Kuhn, and J. Kunkel. A study on data deduplication in hpc storage systems. In *Proc. of the ACM/IEEE Conference on Supercomputing (SC)*, 2012.

[MM03] N. Megiddo and D. Modha. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *Proc. of the USENIX Conference on File and Storage Technologies (FAST)*, 2003.

[MRG+05] P. Maniatis, M. Roussopoulos, T. J. Giuli, D. S. H. Rosenthal, and M. Baker. The lockss peer-to-peer digital preservation system. *ACM Transactions on Computer Systems (TOCS)*, 23(1):2–50, 2005.

[NDR08] D. Narayanan, A. Donnelly, and A. Rowstron. Write off-loading: Practical power management for enterprise storage. *ACM Transactions on Storage (TOS)*, 4(3), 2008.

[PALS09] J.-F. Pâris, A. Amer, D. D. E. Long, and T. Schwarz. Evaluating the Impact of Irrecoverable Read Errors on Disk Array Reliability. In *Proc. of the 15th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC)*, 2009.

[PB04] E. Pinheiro and R. Bianchini. Energy Conservation Techniques for Disk Array-Based Servers. In *Proc. of the 18th Annual International conference on Supercomputing (ICS)*, 2004.

[PBA+05] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. IRON file systems. In *Proc. of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, 2005.

[PGK88] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proc. of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 1988.

[PGM13] J. S. Plank, K. M. Greenan, and E. L. Miller. Screaming Fast Galois Field Arithmetic Using Intel SIMD Extensions. In *Proc. of the 11th USENIX Conference on File and Storage Technologies (FAST)*, 2013.

[Pla08a] J. S. Plank. A New Minimum Density RAID-6 Code with a Word Size of Eight. In *Proc. of the 7th IEEE International Symposium on Network Computing Applications (NCA)*, 2008.

[Pla08b] J. S. Plank. The RAID-6 Liberation Codes. In *Proc. of the 6th USENIX Conference on File and Storage Technologies (FAST)*, 2008.

[PLL13] J.-F. Pâris, D. D. E. Long, and W. Litwin. Three-dimensional redundancy codes for archival storage. In *Proc. of the 21st IEEE International Symposium on Modeling, Analysis, and Simulation (MASCOTS)*, 2013.

[PLS+09] J. S. Plank, J. Luo, C. D. Schuman, L. Xu, and Z. W. O'Hearn. A Performance Evaluation and Examination of Open-Source Erasure Coding Libraries for Storage. In *Proc. of the 7th USENIX Conference on File and Storage Technologies (FAST)*, 2009.

[PRH+12] R. Pitchumani, H. Rekha, A. Hospodor, A. Amer, Y. Kang, E. L. Miller, and D. D. E Long. Emulating a shingled write disk. In *Proc. of the 20th IEEE International Symposium on Modeling, Analysis, and Simulation (MASCOTS)*, 2012.

[PSAL10] J.-F. Pâris, T. J. E. Schwarz, A. Amer, and D. D. E. Long. Improving Disk Array Reliability Through Expedited Scrubbing. In *Proc. of the 5th IEEE International Conference on Networking, Architecture, and Storage (NAS)*, 2010.

[PSAL12] J.-F. Pâris, T. J. E. Schwarz, A. Amer, and D. D. E. Long. Highly reliable two-dimensional RAID arrays for archival storage. In *Proc. of the 31st IEEE Performance Computing and Communications Conference (IPCCC)*, 2012.

[PSLA08] J.-F. Pâris, T. Schwarz, D. D. E. Long, and A. Amer. When MTTDLs Are Not Good Enough: Providing Better Estimates of Disk Array Reliability. In *Proc. of the 7th International Information and Telecommunication Technologies Symposium (I2TS)*, 2008.

[PWB07] E. Pinheiro, W.-D. Weber, and L. A. Barroso. Failure Trends in a Large Disk Drive Population. In *Proc. of the 5th USENIX Conference on File and Storage Technologies (FAST)*, 2007.

[QMW08] J. Qian, C. Meyers, and A. A. Wang. A linux implementation validation of track-aligned extents and track-aligned RAIDs. In *Proc. of the Annual Technical Conference (ATC)*, 2008.

[red] redis. `redis.io`.

[RG10] A. Rajgarhia and A. Gehani. Performance and Extension of User Space File Systems. In *Proc. of the 25th ACM Symposium on Applied Computing (SAC)*, 2010.

[Ros10] D. S. H. Rosenthal. Keeping bits safe: how hard can it be? *Communications of the ACM*, 53(11):47–55, 2010.

[RRM+12] D. S. H. Rosenthal, D. C. Rosenthal, E. L. Miller, I. F. Adams, M. W. Storer, and E. Zadok. The Economics of Long-Term Digital Storage. In *The Memory of the World in the Digital age: Digitization and Preservation.* United Nations Educational, Scientific and Cultural Organization (UNESCO), 2012.

[RS60] I. S. Reed and G. Solomon. Polynomial Codes Over Certain Finite Fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2), 1960.

[RW94] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3), 1994.

[Sag78] C. Sagan. *Murmurs of Earth: The Voyager Interstellar Record.* Ballantine Books, 1978.

[SAP+13] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur. Xoring elephants: Novel erasure codes for big data. In *Proc. of the 39th International conference on Very Large Data Bases (VLDB Endowment)*, 2013.

[SB93] T. J. E. Schwarz and W. A. Burkhard. Multi-Dimensional Disk Array Reliability. Technical report, University of California, 1993.

[SDG10] B. Schroeder, S. Damouras, and P. Gill. Understanding Latent Sector Errors and How to Protect Against Them. In *Proc. of the 8th USENIX Conference on File and Storage Technologies (FAST)*, 2010.

[SG07] B. Schroeder and G. Gibson. Disk Failures in the Real World: What Does an MTTF of 1,000,000 Hours Mean to You? In *Proc. of the 5th USENIX Conference on File and Storage Technologies (FAST)*, 2007.

[SGD+02] S. Saroiu, P. K. Gummadi, R. J. Dunn, S. D. Gribble, and H. M. Levy. An Analysis of Internet Content Delivery Systems. In *Proc. of the 5th Conference on Operating Systems Design and Implementation (OSDI)*, 2002.

[SGLG02] J. Schindler, J. L. Griffin, C. R. Lumb, and G. R. Ganger. Track-aligned Extents: Matching Access Patterns to Disk Drive Characteristics. In *Proc. of the 1st USENIX Conference on File and Storage Technologies (FAST)*, 2002.

[SGMV08] M. W. Storer, K. M. Greenan, E. L. Miller, and K. Voruganti. Pergamum: Replacing Tape with Energy Efficient, Reliable, Disk-Based Archival Storage. In *Proc. of the 6th USENIX Conference on File and Storage Technologies (FAST)*, 2008.

[SHCG94] D. Stodolsky, M. Holland, W. V. Courtright, and G. A. Gibson. Parity logging disk arrays. *ACM Transactions on Computer Systems (TOCS)*, 12(3), 1994.

[SKM12] H. Shim, J. Kim, and S. Maeng. BEST: Best-effort energy saving techniques for NAND flash-based hybrid storage. *IEEE Transactions on Consumer Electronics*, 58(3), 2012.

[SMS+04] J. Satran, K. Meth, C. Sapuntzakis, M. Chadalapaka, and E. Zeidner. Internet Small Computer Systems Interface (iSCSI). RFC 3720, 2004.

[SSP+05] S. W. Schlosser, J. Schindler, S. Papadomanolakis, M. Shao, A. Ailamaki, C. Faloutsos, and G. R. Ganger. On multidimensional data and modern disks. In *Proc. of the 4th USENIX Conference on File and Storage Technologies (FAST)*, 2005.

[STZ10] P. Sehgal, V. Tarasov, and E. Zadok. Evaluating Performance and Energy in File System Server Workloads. In *Proc. of the 8th USENIX Conference on File and Storage Technologies (FAST)*, 2010.

[SXM+04] T. J. E. Schwarz, Q. Xin, E. L. Miller, D. D. E. Long, A. Hospodor, and S. W. Ng. Disk Scrubbing in Large Archival Storage Systems. In *Proc. of the 12th IEEE International Symposium on Modeling, Analysis, and Simulation (MASCOTS)*, 2004.

[TB09] A. Thomasian and M. Blaum. Higher Reliability Redundant Disk Arrays: Organization, Operation, and Coding. *ACM Transactions on Storage (TOS)*, 5, 2009.

[Wal05] C. Walter. Kryder's Law. *Scientific American*, (293), 2005.

[WBM+06] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proc. of the 7th Conference on Operating Systems Design and Implementation (OSDI)*, 2006.

[WGSS96] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID hierarchical storage system. *ACM Transactions on Computer Systems (TOCS)*, 14(1), 1996.

[WM10] A. Wildani and E. L. Miller. Semantic Data Placement for Power Management in Archival Storage. In *Proc. of the 5th Petascale Data Storage Workshop (PDSW)*, 2010.

[WN13] B. Welch and G. Noer. Optimizing a hybrid SSD/HDD HPC storage system based on file size distributions. In *Proc. of the 29th IEEE Conference on Mass Storage Systems and Technologies (MSST)*, 2013.

[WOQ+07] C. Weddle, M. Oldham, J. Qian, A. Wang, P. Reiher, and G. Kuenning. PARAID: A gear-shifting power-aware RAID. *ACM Transactions on Storage (TOS)*, 3(3), 2007.

[Wri12] G. T. Wright. Digital Preservation at the Church of Jesus Christ of Latter-day Saints. Presented at the 28th IEEE Conference on Mass Storage Systems and Technologies (MSST), 2012.

[WT07] M. Woitaszek and H. M. Tufo. Tornado Codes for MAID Archival Storage. In *Proc. of the 24th IEEE Conference on Mass Storage Systems and Technologies (MSST)*, 2007.

[WWKM09] R. Wood, M. Williams, A. Kavcic, and J. Miles. The Feasibility of Magnetic Recording at 10 Terabits Per Square Inch on Conventional Media. *IEEE Transactions on Magnetics*, 45(2):917–923, Feb 2009.

[WZL08] J. Wang, H. Zhu, and D. Li. eRAID: Conserving Energy in Conventional Disk-Based RAID System. *IEEE Transactions on Computers*, 57(3), 2008.

[WZZ+12] J. Wan, N. Zhao, Y. Zhu, J. Wang, Y. Mao, P. Chen, and C. Xie. High Performance and High Capacity Hybrid Shingled-Recording Disk System. In *Proc. of the IEEE International Conference on Cluster Computing (Cluster)*, 2012.

[XMS+03] Q. Xin, E. L. Miller, T. Schwarz, D. D. E. Long, S. A. Brandt, and W. Litwin. Reliability mechanisms for very large storage systems. In *Proc. of the 20th IEEE Conference on Mass Storage Systems and Technologies (MSST)*, 2003.

[XYAZ11] L. Xiao, T. Yu-An, and S. Zhizhuo. Semi-RAID: A Reliable Energy-Aware RAID Data Layout for Sequential Data Access. In *Proc. of the 27th IEEE Conference on Mass Storage Systems and Technologies (MSST)*, 2011.

[YLZ13] Y. Xu Y. Liu, R. Figueiredo and M. Zhao. On the Design and Implementation of a Simulator for Parallel File System Research. In *Proc. of the 28th IEEE Conference on Mass Storage Systems and Technologies (MSST)*, 2013.

[ZCT+05] Q. Zhu, Z. Chen, L. Tan, Y. Zhou, K. Keeton, and J. Wilkes. Hibernator: Helping Disk Arrays Sleep Through the Winter. In *Proc. of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, 2005.

[ZRADAD10] Y. Zhang, A. Rajimwale, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. End-to-end Data Integrity for File Systems: A ZFS Case Study. In *Proc. of the 8th USENIX Conference on File and Storage Technologies (FAST)*, 2010.

[ZSG+03] J. Zedlewski, S. Sobti, N. Garg, F. Zheng, A. Krishnamurthy, and R. Y. Wang. Modeling Hard-Disk Power Consumption. In *Proc. of the 2nd USENIX Conference on File and Storage Technologies (FAST)*, 2003.

List of Figures

2.1 Abstract overview of storage environment. 15
2.2 Histogram of stored ECFS files sizes. 18
2.3 Top: Total number of existing files with amounts of unmodified and unread files since the point in time. Bottom: Fraction of unread or unmodified files relative to existing files. 19
2.4 Total requests per month 21
2.5 Top: CDF over file GET requests per file per size group. Bottom: Zoomed to most frequently retrieved 3 %. 22
2.6 Summed up traffic for GET and PUT requests per unique user id 23
2.7 Mean actions per user sessions for growing window sizes. 24
2.8 Histogram of files sizes 26
2.9 Top: Total number of existing files with amounts of unmodified and unread files since the point in time. Bottom: Fraction of unread or unmodified files relative to existing files. 27
2.10 Throughput and number of accessed fields. 28
2.11 Traffic per unique user id. 28
2.12 Tape states 29
2.13 CDF over load requests per tape cartridge. Left: Absolute. Right: Normalized. 30
2.14 Mount Details for ECFS and MARS 31
2.15 ECFS Cache hit ratio evaluation. 2012 is used for cache warm up. Ratios are measured for 2013/2014. Horizontal red line marks ECMWF's hit ratio. 35
2.16 ECFS Cache hit ratio evaluation. 2012 is used for cache warm up. Ratios are measured for 2013/2014. Horizontal red line marks ECMWF's hit ratio. 36

3.1 Vtrack shifting for $g = 5$ and $s = 3$. 44
3.2 Overview of our evaluation setup. 45
3.3 Energy consumption of HDD and Board (Mainboard + CPU) for defined system states for WD20EARS (R) and WD20EADS (D) running on an Intel D510. 47
3.4 Redundancy groups for on disk formats. 48

3.5 Throughput analysis for in-memory encoding on Intel Atom D510 in logarithmic scale. 49
3.6 Throughput (bottom) and disk energy consumption (top) analysis of full I/O stack for WD20EADS 51
3.7 Throughput (bottom) and disk energy consumption (top) analysis of full I/O stack for WD20EADS 52
3.8 Throughput (bottom) and disk energy consumption (top) analysis of IDR codes defined in Table 3.1 for WD20EADS. The bars represent the codes *A*-*H* (from left to right). 54

4.1 Scrambled on-disk layout and resulting iPGs for $s = 3, in = 2, ik = 1, bs = 4\,\text{kB}$. 62
4.2 LoneStar RAID for d=48 disks with x=8 y=6, dn=32, dk=16, hn=4, hk=1. . . 63
4.3 Pipeline to write *new* data to the RAID. The cache step is optional. 64
4.4 Buckets are accessed via forward-only read- and write promises. 65
4.5 By aggregating writes to overlapping ranges of data disks, the actual work on the parity disks can be reduced. 67
4.6 Examples for irrecoverable error patterns that lead to data loss. All other patterns of simultaneous 4-disk failures can be recovered. 74
4.7 Reliability model for a storage array with non-MDS erasure code, based on Hafner and Rao. 76
4.8 Reads to all disks of a HPG. 81
4.9 Direct writes to random ranges on non-overlapping parity groups. 82
4.10 Direct writes to sequential ranges to all buckets of a HPG. 83
4.11 Delayed writes to random ranges to all buckets of a HPG. 83
4.12 Delayed writes to random ranges to all buckets of a HPG. 32 GB written with 16 GB cache. 84
4.13 Delayed writes to sequential ranges to all buckets of a HPG. 85

5.1 Schematic of an 48 disk LoneStar RAID. X0-X3 denote the buckets with their X+ parity disks of the horizontal parity groups. The bottom row contains the parity disks of the vertical parity groups. 95
5.2 Dependencies during a RAID write. The writes are serialized to overcome the RAID-write hole problem. 97
5.3 Main components of the LoneStar FS. The arrows describe the possible data flows. 97
5.4 On disk format of an extent. 103
5.5 Throughput measurements to quantify the LoneStar FS overhead. Each test was executed at least 10 times and the errors mark a 95% confidence interval. 105

5.6 Throughput of file reads from 6, 12, 18 buckets in parallel. Data is read from disks and checked for integrity. Each test was executed at least 10 times and the errors mark a 95% confidence interval. 106
5.7 24 clients write 64 MB files evenly distributed to 6 buckets to a clean cache. The thin lines of the left axis depict a 10 second rolling mean of the fs / and raid-throughputs, while the thick lines present cache internals. The benchmark stops when all data is fully persisted to the disks. 107
5.8 Raid-throughput during steady state of writing to 1, 2, 4, 6, 8 buckets in parallel with different fixed file sizes. Each test was executed at least 10 times and the errors mark a 95% confidence interval. 108

List of Tables

2.1 File size categorization. Count and capacity refer to Sept. 2014. 17
2.2 Statistics on ECFS tape storage. 18
2.3 Characterization of ECFS workload 2012/01/02 - 2014/05/21. 20
2.4 ECFS user session analysis. A total of 2,709,343 sessions were identified. . 23
2.5 Statistics on MARS' tape storage. 25
2.6 Characterization of MARS workload 2010/01/01-2014/02/27. 29
2.7 Tape mount statistics 30
2.8 Prefetching hits based on the correlation analysis 32

3.1 Codes under test with parameterization $m + k$, scrambling s, resulting storage efficiency $1 - r$ and resulting ePG sizes for IDR code block sizes 512 byte and 4 kB. 53

4.1 Model parameters for MTTDL analysis 75
4.2 MTTDL in hours for 10 PB user data built with multiple arrays of $n + k$ RAID schemes. 77
4.3 MTTDL in hours for 10 PB user data built with multiple LoneStar arrays. . 78
4.4 Feasible configurations for a 192 disks system. All disks use a fixed 16+1 SPC code for intra-disk redundancy. 79
4.5 Ratio of effective throughput to sum of all disks' runtimes. Metric is equivalent to $\frac{MB}{Joule}$. 85
4.6 Ratio of effective throughput to sum of all disks' runtimes. Metric is equivalent to $\frac{MB}{Joule}$. 86