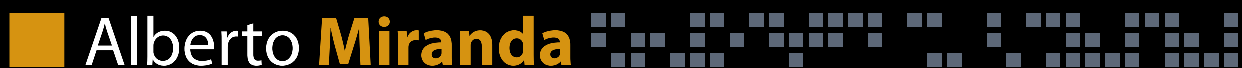
SCALABILITY **IN EXTENSIBLE** AND **HETEROGENEOUS** STORAGE SYSTEMS

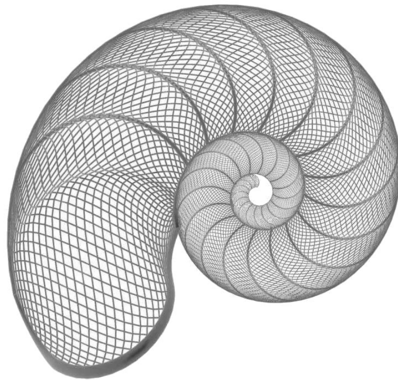
Alberto **Miranda**



SCALABILITY in extensible *and heterogeneous* STORAGE SYSTEMS

SCALABILITY in extensible *and heterogeneous* STORAGE SYSTEMS

Alberto Miranda



Scalability in Extensible and Heterogeneous Storage Systems
by Alberto Miranda Bueno

A thesis submitted in partial fulfillment of the requirements for the degree of *Doctor of Philosophy in Computer Science*.
Copyright © 2009–2014 Alberto Miranda Bueno.
All rights reserved.
Printed in Spain.

01 12 35 81 32 13 45 58 91 44
First edition: September 2013
Second edition: June 2014

Advisor:	Prof. Dr. Toni CORTÉS ROSSELLÓ
Department:	Department of Computer Architecture (DAC)
University:	Universitat Politècnica de Catalunya–BarcelonaTech (UPC)
Pre-dissertation Committee:	Prof. Dr. Julita CORBALÁN GONZÁLEZ Prof. Dr. Jordi TORRES VIÑALS Prof. Dr. Juan José COSTA PRATS
Dissertation Committee:	Prof. Dr. Angelos BILAS Prof. Dr. Pilar GONZÁLEZ-FÉREZ Prof. Dr. Julita CORBALÁN GONZÁLEZ

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks™ or registered® trademarks. Where those designations appear in this book, and the author was aware of a trademark claim, the designations have been printed in caps or initial caps, and/or with the corresponding symbols.

This work was partially supported by the Spanish Ministry of Economy and Competitiveness under grants TIN2007-60625, BES-2008-006019, SEV-2011-00067 (Severo Ochoa Program) and TIN2012-34557 and by the Catalan Government under grant 2009-SGR-980. Part of the research was also supported by the European Community under the Marie Curie Initial Training Network SCALUS grant agreement #238808 and the Seventh Framework Programme (FP7/2007-2013) under grant agreement RI-283493, as part of the PRACE-2IP project.

Departament d'Arquitectura de Computadors
UNIVERSITAT POLITÈCNICA DE CATALUNYA

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

en memoria de mi madre
Isabel Bueno Sáez
31/12/1952–14/11/2013

Abstract

The evolution of computer systems has brought forth an exponential growth in data volumes, which is pushing the capabilities of current storage architectures to organize and access this information effectively: as the unending creation and demand of computer-generated data grows at an estimated rate of 40–60% *per year*, storage infrastructures need increasingly scalable data distribution strategies that are able to adapt to this growth and provide adequate performance.

Hardware-wise, the most flexible approach consists in using pools of storage devices that can be expanded as needed by adding new devices or replacing older ones, thus seamlessly increasing the system's performance and capacity. Such an approach, however, necessitates data distribution strategies that can adapt to these changes in the infrastructure and can also exploit the potential performance offered by the hardware. Such strategies should be able to rebuild the data layout to accommodate the new devices in the storage infrastructure, extracting the utmost performance from the hardware and offering a balanced workload distribution. An inadequate data layout might not effectively use the enlarged capacity or better performance provided by newer devices, thus leading to unbalancing problems like bottlenecks or resource underutilization.

The title of this dissertation, 'Scalability in Extensible and Heterogeneous Storage Systems', refers to the main focus of our research in scalable data distributions that can adapt to increasing volumes of data. With this thesis we make several novel contributions to storage research: first we design and evaluate a pseudo-randomized distribution strategy that can adapt to hardware changes while redistributing only the minimum data to keep a balanced workload; second, we perform a comparative study about the influence of pseudo-random number generators on the performance and distribution quality of randomized distributions; third, we conduct an analysis of long-term data access patterns in several real-world traces in order to determine if it is possible to offer high performance and a balanced load with less than minimal data rebalancing; fourth, we apply the knowledge learned on long-term access patterns to design an extensible RAID architecture that can adapt to changes in the number of disks without migrating large amounts of data.

Contents

Abstract ix

List of Figures xv

List of Tables xvii

Preface xix

0 Humans and information storage 1
- 0.1 Overview and objectives 5
- 0.2 Publications 7
- 0.3 Finding your way around 8

1 Background and related work 11
- 1.1 Distributed file systems background 11
- 1.2 Storage architectures 13
- 1.3 RAID 14
- 1.4 Object-based storage 18
- 1.5 Data distribution, reliability and adaptivity 19
- 1.6 Hierarchical storage and layout optimization 24
- 1.7 Characterization of storage behavior 25

2 Scalable data distribution 29

2.1 Motivation 29
2.2 Research model 32
2.3 Randomized data distribution 33
2.4 Proposal: Random Slicing 37
2.5 Methodology 43
2.6 Scalability of Random Slicing 44
2.7 Comparative evaluation 51
2.8 Conclusions 63

3 PRNGs in data distribution 65

3.1 Motivation 65
3.2 Introduction to (pseudo-)randomness 67
3.3 Methodology 70
3.4 Influence on fairness 72
3.5 Influence on performance 77
3.6 Evaluation summary 80
3.7 Conclusions 80

4 Long-term locality in mass storage 83

4.1 Motivation 83
4.2 Methodology 85
4.3 Block sharing 88
4.4 Block usage 92
4.5 Scope of our hypothesis 99
4.6 Conclusions 103

5 Extensibility in RAID architectures 107

5.1 Motivation 107
5.2 Proposal: CRAID 110
5.3 Methodology 115
5.4 Management of the Cache Partition 121
5.5 CRAID-5* Response Time 122
5.6 CRAID-5* Workload Distribution 127
5.7 Performance of CRAID-0* 129
5.8 Conclusions 132

6 Conclusion 135

A The ENT test for pseudo-random sequences 141

B The NIST test suite 145

B.1 Frequency (MONOBIT) test 146
B.2 Frequency test within a block 146
B.3 Runs test 147
B.4 Test for the longest run of ones in a block 148
B.5 Binary matrix rank test 148
B.6 Discrete Fourier transform (SPECTRAL) test 149
B.7 Non-overlapping template matching test 149
B.8 Overlapping template matching test 150
B.9 Maurer's "Universal Statistical" test 151
B.10 Linear complexity test 152
B.11 Serial test 152
B.12 Approximate entropy test 153
B.13 Cumulative sums (CUSUM) test 154
B.14 Random excursions test 154
B.15 Random excursions variant test 155

Bibliography 157

Further Reading 173

Index 177

List of Acronyms 183

List of Figures

0.1 The digital universe: 50x growth from 2010 to 2020 2
0.2 CAGR trends in disk drive technology 3

1.1 Striping with parity 15

2.1 Overview of Random Slicing's data distribution 38
2.2 Example of the CutShift+Sorted algorithm 42
2.3 Percentage of new intervals created 46
2.4 Total number of intervals created 47
2.5 Lookup time after several reorganizations 50
2.6 Scalability of fairness in homogeneous settings 52
2.7 Impact of the number of points on Consistent Hashing's fairness 54
2.8 Fairness of Share depending on the stretch factor 55
2.9 Scalability of fairness in heterogeneous settings 56
2.10 Memory usage and performance in heterogeneous settings 58
2.11 Scalability of adaptivity in heterogeneous settings 59
2.12 Fairness of Consistent Hashing using a fixed number of points 62

3.1 Three-dimensional plot of the first numbers generated by RANDU 68
3.2 Influence of PRNG subclasses on fairness 73
3.3 Existence of patterns in the first numbers generated by several PRNGs 75
3.4 Influence of PRNG subclasses on performance 78

4.1 Time required for data migration due to capacity upgrades 84
4.2 Individual and binned percentages of shared blocks 88
4.3 Working set overlap and time in general purpose workloads 90
4.4 Working set overlap and time in specialized workloads 91
4.5 Individual and binned percentages of accesses to shared blocks 93
4.6 CDF of blocks' lifespan by block count 94
4.7 CDF of blocks actual usage by block count 96
4.8 CDF of consecutive usage by block count 97
4.9 Daily count of blocks with 90% accesses 99
4.10 Block-frequency in the examined traces 101
4.11 Working-set overlap in the examined traces 102

5.1 RAID restriping process 108
5.2 CRAID architecture using RAID-0 112
5.3 Software components and I/O control flow of CRAID 113
5.4 Overview of configurations based on RAID-5 118
5.5 Overview of configurations based on RAID-0 119
5.6 Read response time in CRAID-5* 123
5.7 Sequentiality CDFs for traces cello99 and webusers 124
5.8 Write response time in CRAID-5* 126
5.9 CDFs of the coefficient of variation in traces deasna and wdev 128
5.10 Average I/O response time in CRAID-0* 130
5.11 CDFs of c_v for traces cello99, deasna, home02 and webusers 131

List of Tables

2.1 Pros and cons of several approaches to data distribution 31
2.2 Properties of examined strategies in heterogeneous architectures 63

3.1 List of evaluated PRNGs 71
3.2 Results obtained with the ENT suite 76
3.3 Results obtained with the NIST suite 77
3.4 Final PRNG ranking 79

4.1 Summary of traces examined 86
4.2 Summary statistics of traces from seven different systems 100

5.1 Hit ratio for each cache partition management algorithm 121
5.2 Replacement ratio for each cache partition management algorithm 122
5.3 Best hit ratio and worst eviction ratio from all simulations 124
5.4 Comparison of CRAID's dedicated vs. non-dedicated approach 125
5.5 Influence of P_C size on workload distribution 129

Preface

Dear prospective reader,

Research can be daunting sometimes, but it is certainly an enjoyable experience. When I first met with my advisor, Toni Cortés, a number of years ago to discuss what being a researcher was like, he replied "It is akin to playing with the largest, funniest toys in the world." And boy, was he right; during my research I re-experienced the childhood joy (and frustration) of trying to build whatever my imagination came up with, feeling certainly stupid at times whenever I was incapable of seeing apparently obvious solutions, and wickedly clever whenever I felt that my contributions had led to something really original. Whatever the case, this thesis is the extensive result of the bubbling ideas from my slightly deranged mind, and represents a formal way of asking society whether it deems that my contributions are really innovative and original.

In my opinion, even though a thesis is necessarily a work by one author, more often than not it would be impossible without the selfless collaboration of a certain number of people. This is my, admittedly small, recognition to all those who have helped (voluntarily or not) during my research process.

First of all, I owe a deep professional and personal gratitude to my advisor, Prof. Dr. Toni Cortés. On a professional level, he helped me mature as a computer engineer (by being one of my professors and also the advisor of no less than three master theses) and as a researcher (by offering his guidance and unwavering confidence in that everything works out in the end). I am particularly fond of those

discussions where we contradict each other constantly in order to eventually shape an idea, and I am sorry if could not write this thesis in "one afternoon". On a personal level, I cannot describe how grateful I am for his support during some of the worst moments in my life.

Besides my advisor, I very much appreciate the collaboration of all the (current and past) members of the STORAGE SYSTEMS RESEARCH GROUP, specially of Dr. Ernest Artiaga who has an uncanny ability to detect and destroy weak argumentations. With him I have really learned to think things through before proposing them aloud.

I must not forget to mention my gratitude to Prof. Dr. André Brinkmann from the Johannes Gutenberg–Universität Mainz, who has helped several times to improve the quality of the research presented in this dissertation, and could be considered like a "foster advisor" of sorts.

Part of the research in this thesis would not have been possible without the high-performance computing infrastructures kindly provided by the Barcelona Supercomputing Center (BSC–CNS) and the Department of Computer Architecture at the Universitat Politècnica de Catalunya–BarcelonaTech (UPC). To them, and specially to the people in their SYSTEMS and SUPPORT departments I owe my most sincere gratitude.

Writing this thesis has also offered me the opportunity to dabble into what has turned to be a really interesting craft: typography and the typesetting of long text documents. While doing this, I have learned many things about bookmaking, specially from the always helpful people at *TEX StackExchange*, and I have tried to apply it to the document you are now holding in your hands. Every decision I took was always made with you, the reader, in mind: it was always my intention to improve the legibility of the content and increase the comprehensiveness of the information presented; the result is now for you to judge.

Besides the people directly related with my research, I must also thank the members of the PROGRAMMING MODELS RESEARCH GROUP Xavier Teruel, Javier Bueno, Roger Ferrer and Juan José Costa for creating a cordial and fun workplace from day one. I must not forget Dr. David Ródenas, Jordi Vaquero, Miguel Tomás and Jairo Balart (whose annoying coffee jokes are certainly exhausting) for sharing a whole lot of good times.

I also need to thank my friends Iván García, Jorge Sánchez, Toni Molina and, most specially, Alejandro Touza for being with me when I needed them most. Even when our lives make it difficult to meet often, you are still the greatest friends!

Para concluir, debo darle las gracias a mi familia porque sin su apoyo y sacrificio nunca habría llegado hasta aquí. A mis padres Isabel y Antonio, que trabajaron muy duro durante muchos años para que mi hermano y yo tuviéramos oportunidades que ellos nunca tuvieron. Gracias por demostrarme que el amor incondicional existe aún en los peores momentos. A mi hermano José Andrés, por creer en mí como deben hacer todos los hermanos mayores, y especialmente por escribir aquella carta de despedida cuando yo no era capaz. A mi sobrina Aina, por iluminar la vida con su sonrisa cuando es más necesario. A Lidia, mi mujer, por estar siempre ahí; por tu paciencia, tu soporte y por hacer mi vida un poco más feliz cada día.

A todos vosotros os dedico este trabajo, pero en especial te lo dedico a ti, mamá, después de los malos momentos y los años de lucha te merecías más que nadie poder vivir para leer esto y asistir a la lectura. Espero que estés orgullosa y ojalá volvamos a vernos. Hasta siempre.

Barcelona, 15 June 2014

A. Miranda

Chapter 0

Humans and Information Storage

"Begin at the beginning," the King said, gravely, "and go on till you come to an end; then stop."
— Lewis Carroll, *Alice in Wonderland*

"He'd always felt he had a right to exist as a wizard in the same way that you couldn't do proper maths without the number 0, which wasn't a number at all but, if it went away, would leave a lot of larger numbers looking bloody stupid."
— Sir Terry Pratchett, *Interesting Times*

Information storage has accompanied mankind for at least 40,000 years. Before the Sumerian writing system—which is considered the first writing script ever developed—was invented *ca.* 3200 BCE, accounting and record keeping were practiced by carving tally marks in wood, bone, and stone [67]. Later, the invention of more sophisticated writing materials like papyrus, leather, parchment and, finally, wood-pulp paper (China *ca.* 100 CE) dramatically facilitated reading and writing and greatly increased the density of information conveyed in that early, rudimentary media.

After the invention of modern paper and the printing press, came a steady decline in the manufacturing and acquisition cost of writing material, which further increased the production of information. Nevertheless, it was not until the popularization of electronic computers and digitally aware personal devices that the means to easily produce and process large amounts of data were readily available.

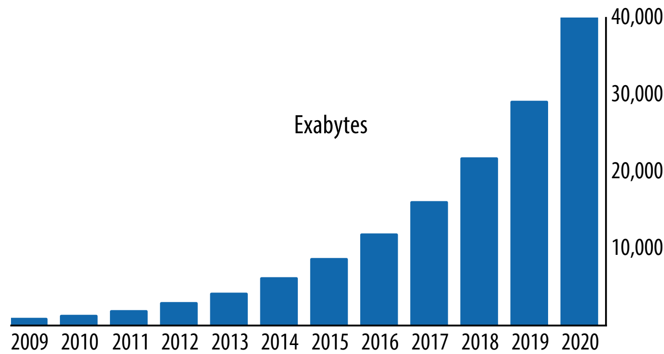


Figure 0.1: The digital universe. Historical and predicted growth of stored digital data due to the aggregation of social, mobile, cloud and analytic sources. Source: Gantz and Reinsel. "The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east" [47].

Today, the amount of information created by each individual—be it documents written, pictures taken or music downloaded—is much larger than anything we have ever seen before.

As a result, digital data volumes are growing exponentially (see Figure 0.1↑). The amount of online, digital data exceeded 1.8 Zettabytes in 2011 and is estimated to continue to grow at a 40–60% compound annual growth rate (CAGR), leading to a projected output of 7.9 Zettabytes for 2015 [46, 47]. If this growth continues to hold true, an enterprise with a respectable 200 TB of stored data at the end of 2013 will have more than 1.6 PB of data by 2019, and we are not considering the extra information required for backing up or replicating that data.

In order to provide the required performance and reliability, large-scale storage systems have traditionally relied on multiple RAID-5 or RAID-6 configurations of storage devices, interconnected with high-speed networks like Fibre Channel or SAS.[1] Unfortunately, the performance of the current, most commonly-used storage technology—the magnetic disk drive—is unable to keep pace with the rate of growth needed to sustain the aforementioned data explosion. Though the current growth rate of areal density (GB/in^2) of disks is at 25–40% per year (see Figure 0.2a↷), faster access times are limited by improvements on both rotational latency and seek time. Rotational latency has settled down to 2, 3, and 4.1 milliseconds, depending on whether the disk spins at 7200, 10,000 or 15,000 revolutions per minute, respectively, and faster speeds are not expected within the next 5 years, if at all [45]. The other component of access time, seek time, is not expected to im-

[1] *Serial Attached SCSI.*

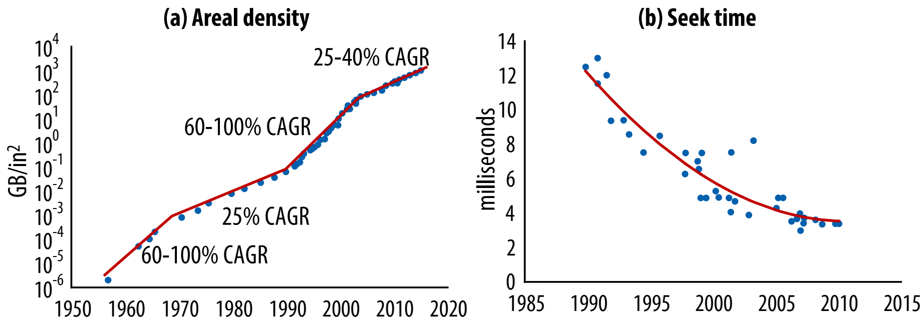


Figure 0.2: CAGR trends in disk drive technology. Figures show the historical and predicted evolution of disk drives areal density and seek time. Source: Freitas, Slember, Sawdon, and Chiu. "GPFS Scans 10 Billion Files in 43 Minutes" [44].

prove much. Historically, its improvement has been less than 5% per year and there are no indications that this trend will change in the future (see Figure 0.2b↑).

On the other hand, storage architectures based on solid-state devices (SSDs), which are the theoretical successors of current magnetic drives, do not seem prone to replace HDD-based storage for the next 5 to 10 years, at least in data centers. Even though the performance of SSDs significantly improves that of hard drives, the main problem resides in the lack of manufacturing capacity. While in 2012 the HDD industry delivered 66,358 PB in 63.4 million enterprise-grade hard-drives, the NAND industry delivered only 1781 PB in 5.7 million SSDs. It would cost the NAND industry hundreds of billions of dollars to build enough manufacturing plants to satisfy the forecasted demand for enterprise storage [48].

Conversely, improvements in magnetic recording could improve the capabilities of hard drives significantly. For instance, the emerging *shingled magnetic recording* achieves higher densities and performance by removing the gaps between disk tracks,[2] and could be deployed with existing manufacturing lines. In addition, development of other technologies like *heat-assisted magnetic recording* [119], *bit-patterned recording* [116, 112] and CPP/GMR[3] heads [35], is underway to maintain the competitiveness of hard-disk drives with SSDs, and could topple the apparent advantage of solid-state technology. In any case, however, it is still not clear if the improvements derived from these technologies will be enough to sustain the upcoming data growth.

[2] *An HDD track is the circular path on the surface of a disk where data is magnetically recorded.*

[3] *Current Perpendicular to Plane Giant Magnetoresistance.*

Besides the problems derived from technological and mechanical limitations, the massive amounts of data growth impose more challenges. First, large-scale storage systems are required to expand according to the needs of the users, and they must do so in an efficient manner. This means, for instance, that whenever new devices are installed, large amounts of data must be migrated from the old storage devices to the new, in order to keep a balanced data load and use the new devices appropriately. The reason for this, is that an inadequate balance of the data workload can lead to misused hardware, overloading devices with too much data or client requests and underusing others. Nevertheless, in the case of Petascale storage systems, the amounts of data to be moved to regain a balanced workload can be enormous, and the required migration times cost-prohibitive.

Second, massive storage systems will inevitably be composed of a collection of heterogeneous hardware: as capacity and performance requirements grow, new storage devices must be added to cope with demand, but it is unlikely that these devices will have the same capacity and/or performance that those currently installed. Furthermore, upon failure, disks are most commonly replaced by faster and larger ones, since it is not always easy (or cheap) to find a particular model of drive. In the long run, any large-scale storage system will have to cope with a myriad of devices (SATA/SATA2/SATA3 or SCSI hard drives, magnetic tapes, SSDs, …) with very different performance characteristics and capacities, and should be able to aggregate and exploit them adequately.

To address these challenges, in this dissertation we propose two novel mechanisms, based on several of our contributions, that improve the resilience of large-scale storage against huge volumes of data. The first mechanism, called Random Slicing, uses a controlled pseudo-randomized data distribution in order to adapt to changes in the storage architecture. This permits to migrate only the minimum amount of data required to keep a balanced I/O distribution, while easily supporting the hardware heterogeneity of the devices.

The second mechanism, called CRAID, is a RAID extension that optimizes the layout of long-term, frequently-used data with the goal of maintaining a good performance when extending the array of devices. By identifying and tracking changes to this data subset, the mechanism is able to distribute only relevant data into the new disks in an on-line fashion, which significantly reduces the amount of work necessary to use the new disks in a balanced manner.

0.1 Overview and Objectives

The main achievement of this thesis consists in proving that, by tracking long-term data locality and optimizing its on-storage layout, it is possible to improve the performance of mass storage systems and have the ability to gracefully adapt to dynamic changes in the underlying architecture, without a loss in efficiency. In order to reach this conclusion, our research has made several novel contributions that originated from the following three main research objectives.

1. *Determine scalability issues.* As we have already discussed, the current approaches to upgrade heterogeneous storage systems are insufficient to handle data loads in the Petascale order of magnitude or beyond. The main cause for this problem, is either because data rebalancing takes too long to perform or because the internal data structures used to speed up this process occupy too much memory. The allocation policies derived from our research should be able to organize data in a way that favors a graceful redistribution to new disks, without requiring neither excessive time nor memory.

2. *Achieve good storage performance with minimal data migration.* Once the issues that affect scalability are determined, we need to be able to design a scalable distribution strategy that requires only minimal data migration. Notice that by minimal data migration we refer to the *minimum amount of data required to use the storage devices effectively*. As we will see, some strategies already exist, but they can be improved either in terms of memory consumption or response time.

3. *Achieve good storage performance with less than minimal data migration.* Minimal data migration may not be enough in Petascale or Exascale storage systems. For this reason, scalable solutions are needed that offer good performance levels with even less data redistribution. To achieve this goal, we focus on using the properties of long-term, heavily accessed data in an attempt to reduce the amount of work needed. In order to achieve this goal, we need to prove the existence of long-term data locality, demonstrate that long-term data can be predicted with high reliability, and verify the technical feasibility of locality-based layout optimizations.

Although this overview is necessarily linear, these goals are interspersed between the different chapters of the document; therefore, let us first have a brief look at *what* each chapter contains and *why*. Note that we disregard this very same '*Humans and information storage*'$_{1}$ and the '*Conclusion*'$_{135}$ from the upcoming description, since their contents are pretty much self-explanatory.

0.1.1 Background and Related Work

To make sure that we start from common ground, Chapter 1, '*Background and related work*'$_{11}$, gives an overview of the basic concepts needed to understand this dissertation and reviews the existing literature on the field. As such, a good understanding of its contents is essential prior to reading any of the other parts. The main topics of this chapter include key concepts about current storage architectures, data distribution and hybrid/hierarchical storage devices. We also discuss some of the design issues involved in the conception of large-scale storage systems, and the new interfaces to access stored data.

0.1.2 Scalable Data Distribution

Chapter 2, '*Scalable data distribution*'$_{29}$, presents the design and experimental evaluation of the Random Slicing data distribution strategy. We also discuss the theoretical evaluation model on which pseudo-randomized data distribution strategies are based, and we perform a comparative evaluation of our proposal against some of its best well-known competitors. With this chapter, we demonstrate that it is possible to design a highly-scalable data distribution with minimal data migration, with low memory and performance overheads.

0.1.3 PRNGs in Data Distribution

In Chapter 3, '*PRNGs in data distribution*'$_{65}$, we discuss the issues associated to using pseudo-randomized number generators for data distribution. In an attempt to make it as self-contained as possible, we provide a bit of the relevant background on the design and implementation of random number generators, and also evaluate their influence on all the distribution strategies discussed in Chapter 2. The main contribution of this chapter is a ranking of the best pseudo-random number generators, both in terms of data distribution balancing and performance.

0.1.4 Long-term Locality in Mass Storage

With Chapter 4, '*Long-term locality in mass storage*'$_{83}$, we dive into the semantics of data by examining the access patterns in large-scale storage systems. We analyze in detail twelve storage traces focusing in the clients' long-term access patterns and locality. Notice that though the presence of short-term data locality has been extensively demonstrated, there is nowhere near as much literature regarding long-term data locality. In addition, since the main goal of the chapter is to identify general access traits that can be exploited to improve the scalability of current storage systems, the traces discussed represent a variety of different workloads and were captured at different points in time over the last 14 years.

More specifically, in this chapter we show: (1) that there is an important amount of long-term locality in mass storage systems, and that it makes sense to apply specific optimizations for it; (2) that this subset of "reused" data is small (compared to the overall dataset), and that there are benefits in rebalancing it instead of the whole dataset when upgrading the mass storage system; (3) that this subset is not volatile, that is, that any changes to it happen over a period of time long enough to compensate the time spent optimizing its on-device location.

0.1.5 Extensibility in RAID Architectures

Finally, in Chapter 5, '*Extensibility in RAID architectures*'$_{107}$, we use the information learned about access patterns in Chapter 4 to prove that it is possible to extend current RAID architectures by redistributing only frequently accessed data, without a significant loss in performance. We discuss the design of CRAID and also present its simulation-driven, experimental evaluation, with special emphasis on comparing it against current RAID solutions.

0.2 Publications

This dissertation is the amalgamation of several research ideas that we have developed over four years of research. That being the case, some of the ideas and figures presented in this dissertation have appeared previously (or are bound to appear) in several peer-reviewed international conferences, a scientific journal and a technical deliverable for an EU project (which includes our research on extensible data layout optimizations for RAID-0 arrays).

The following list shows the bibliographic references of all these scientific works, with entries sorted in reverse chronological order of appearance:

- [**Miranda2014b**] A. Miranda, S. Effert, Y. Kang, I. Popov, T. Friedetzky, E.L. Miller, A. Brinkmann, and T. Cortes. “Random Slicing: Efficient and Scalable Data Placement for Large-scale Storage Systems”. *ACM Transactions on Storage* 10.3 (2014), 36.

- [**Miranda2014a**] A. Miranda and T. Cortes. “CRAID: Online RAID Upgrades Using Dynamic Hot Data Reorganization”. *Proceedings of the 12th USENIX Conference on File and Storage Technologies, 2014*. Santa Clara, CA: USENIX, Feb. 2014, 133–146. isbn: 978-1-931971-08-9.

- [**Artiaga2013**] E. Artiaga and A. Miranda. “PRACE-2IP Technical Deliverable D12.4: Performance Optimized Lustre”. *INFRA-2011-2.3.5 – Second Implementation Phase of the European High Performance Computing (HPC) service PRACE* (2012).

- [**Miranda2012**] A. Miranda and T. Cortes. “Analyzing Long-Term Access Locality to Find Ways to Improve Distributed Storage Systems”. *20th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, 2012*. IEEE. Garching, Germany, Feb. 2012, 544–553. doi: 10.1109/PDP.2012.15.

- [**Miranda2011**] 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”. *Proceedings of the 18th International Conference on High Performance Computing, 2011*. IEEE. Bangalore, India, Dec. 2011, 1–10. doi: 10.1109/HiPC.2011.6152745.

0.3 Finding Your Way Around

This thesis, as most scientific works, contains a lot of both internal and external references. The internal references consist of chapters, sections, subsections and (a few) theorems and equations. The electronic version of this document is fully hyperlinked, and it is easy to go back and forth through the relevant contents. Since it is impossible to include such commodities in the “dead tree edition” of this work,

all the cross-references that are located on the same double-page spread are accompanied by a visual cue to look above (↑), at the recto (↷) page or at the verso (↶) page, depending on where the referenced content is located.[4] For cross-references that are not located in the same page spread, we include the relevant page number as a subindex to the reference.

[4] *Also, any side remarks or concepts will be done using marginal notes.*

In addition, to allow this thesis to be used as a reference itself, we include an Index$_{177}$ of the most relevant terms and, additionally, a plain List of Acronyms$_{183}$ as well. Bibliographic data about external references is collected in the Bibliography$_{157}$ and the Further Reading$_{173}$ sections, which include, respectively, texts directly cited in the dissertation and additional works worth reading.

Chapter 1

Background and Related Work

"Student: Dr. Einstein, aren't these the same questions as last year's [physics] final exam?

Dr. Einstein: Yes; But this year the answers are different."
— Albert Einstein

As we have seen, systems with billions of files and Petabytes of storage are slowly becoming more and more common [15]. In this scenario, the constraints placed on modern storage systems are tough: they must be able to manage huge amounts of data, provide efficient search mechanisms and organize millions of concurrent requests to minimize competition issues. Furthermore, they must be elastic enough to grow when new devices are installed and react appropriately when those devices malfunction or need to be removed because of old age.

There is a wide range of solutions taking on these problems, though nowadays, with the advent of cloud computing, there is a special interest on distributed environments. In the following sections we will discuss the most relevant approaches to the problems related to high performing Petascale and Exascale storage.

1.1 Distributed File Systems Background

A file system is usually described as a subsystem of an operating system whose purpose is to provide long-term storage [79]. To achieve this goal, the file system produces *files*: named objects created to store user data that exist from their explicit

creation to their explicit destruction. By keeping these files into secondary storage devices (i.e. magnetic disks, tapes or SSDs) the file system guarantees that this data will not be erased when the system goes offline, and also (relatively) protects it from system failures. The file system also provides a set of *file operations* that its clients can use to operate on files: *create* a file, *delete* a file, *read* from a file and *write* to a file are the most common ones.

By extension, the purpose of a distributed file system is to allow users of physically distributed computers to share data. By its very own nature, a distributed file system will be run on a set of dispersed machines interconnected by a network but, ideally, it should appear to its clients as a conventional, centralized file system. It does not matter if those clients are human beings using a workstation or a software component running on a server, as long as the file system is able to provide a unified, consistent view of the data. Distributed file systems achieve this goal by means of independent software components that cooperate with each other to provide a common service interface. This interface provides the implementation of the file operations discussed above, effectively abstracting processes from the fact that they are running on a distributed environment. This *component oriented* approach, however, means that bookkeeping information and file data have to be transported across a network, therefore incurring an overhead that classical centralized file systems do not have.

The software components that build up a distributed file system can be roughly classified as *clients*, *file servers* and *storage servers* (although, of course, particular implementations may vary). *Clients* capture the file operation requests issued by file system users and forward them to the appropriate components distributed throughout the system. They are the software layer that provides the file operation services and, therefore, the facade that hides the complexities of the system from users. *File servers* index files, properly organize data and manage all the administrative information—*metadata*—referred to them (such as owner, size, access permissions and timestamps). *Storage servers*, on the other hand, control physical storage devices and the way users data is stored and retrieved. They are also often in charge of making the necessary adjustments when new devices are added to the system or disappear from it (due to removal or failure).

In the context of this dissertation, we will be focusing mainly on strategies for distributed storage servers, and from time to time we will discuss the facilities that need to be implemented in clients (and how they need to coordinate among each other) in order to successfully carry them out.

1.2 Storage Architectures

Choosing the appropriate architecture to access data from multiple locations is a major concern for current distributed systems. Ideally, a storage architecture should provide a strong security infrastructure, seamless data sharing across different platforms, high performance and scalability in terms of clients and devices. There are three main storage architecture in use nowadays, wich we review in the following paragraphs: *direct-attached storage* (DAS), *storage area networks* (SAN) and *network-attached storage* (NAS).

A DAS architecture connects block-based storage devices to the I/O bus of a server or workstation (usually via SCSI, SAS or Fibre Channel), without the need for a network in between. It offers high performance due to the high data bandwidth and access rate offered by this direct connection, as well as minimal security concerns. Nevertheless, there are limits to the number of devices than can be directly connected (e.g. a SCSI 16-bit bus will support at most 16 hosts or devices) and it is not possible to access the contents of such devices from another host.

SAN architectures were developed to address the connectivity limitations of DAS. SAN architectures use a fast and scalable interconnect network (such as iSCSI or Fibre Channel) to connect remote storage devices to servers in a way that they appear to be locally attached. For instance, in the case of iSCSI, each set of blocks is exposed as an *iSCSI target* and can be mounted by an *iSCSI initiator*, a setup that has been proven to be competitive in terms of performance [4]. However, since a SAN can be shared by different machines, new security concerns appear which require for the introduction of new concepts like host-device authentication or zoning [136].

In DAS and SAN architectures, the file system is responsible for mapping the blocks in the different devices with its high level data structures (files and directories). When access is shared among several hosts, this information must also be shared and kept consistent across all the hosts. The complexity of this task is one of the major drawbacks that effectively limits scalability in terms of number of hosts.

The most common approach used to improve this scalability on the host side, is to use a NAS architecture. In a NAS, there is a reduced number of hosts acting as dedicated file servers that are directly attached to storage devices using a SAN architecture, while the rest of the clients access the storage system indirectly using such file servers. Hence, metadata is completely managed by the file servers, and

its reliability and consistency is independent of the number of clients. The NFS file system [27, 130] is a clear example of such an architecture.

NAS architectures shift the pressure of scalability from the storage devices to the file servers, which have to channel all the storage traffic and can become a bottleneck. SAN filesystems were introduced in order to mitigate this effect. In a SAN file system, both file servers and clients are connected to a SAN: file servers can now share metadata with its clients and then recover the data directly from the devices. This way, consistency is guaranteed by a small number of servers, without hindering the I/O bandwidth. Security is the main drawback of a SAN file system, however, because devices export data as fixed-sized blocks with no information about their contents. Since all hosts have access to all storage devices, there is no mechanism to validate I/O operations and prevent a malicious client from altering the data stored. Due to this problem, some file systems add additional mechanisms like encryption or digital signatures (Farsite [1]) or even replace block-based devices with Object Storage Devices (OSDs),[1] which can enrich data with semantic information (zFS [117], Ceph [146], Lustre [18]).

[1] *We will review OSDs in detail in Section 1.4**18**.*

Now that we know how to access data stored in multiple networked locations, let us review the more common, high-performing technologies used for storing data in physical devices.

1.3 RAID

The term RAID [109]—an acronym for Redundant Array of Independent Disks—describes a technology to distribute data across several physical disks with the purpose of improving performance and reliability. This is done by providing a controller, which can be implemented in hardware or software, that groups disks into a single logical storage device and handles all data transfers to it by redistributing them among the original disks. Even though this involves adding significant computation when reading and writing data, in deployments where high-performance is a concern, a hardware controller can do the work efficiently.

Three are the key mechanisms on which RAID technology is based: *mirroring*, *striping* and *error correction*, with the first two being mutually exclusive.[2] A particular chunk of data is said to be mirrored if several copies of it are sent to more than one disk, whereas it is said to be striped if it is sliced into fragments—*striping units*—that are sent to different disks (see Figure 1.1↷). With error correction, parity information is stored in order to detect and repair data consistency problems.

[2] *A RAID volume can choose to use either mirroring or striping, but not both.*

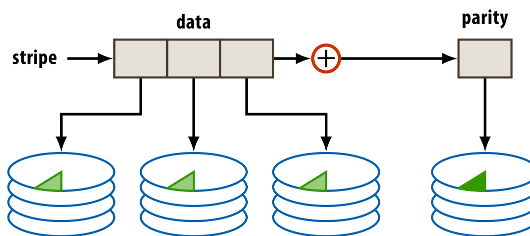


Figure 1.1: Striping with parity. The storage space of a RAID disk array is divided into stripes, where each stripe contains one striping unit on each disk of the array. One of the striping units holds parity data that can be used to recover after a disk failure.

Notice that the impact these mechanisms have in a RAID configuration is significant: basic mirroring can speed up reading as the system can acquire the data from more than one disk at once, but penalizes writing as all the disks have to confirm that the data was correctly written; striping is often used for performance since it allows sequences of data to be read from multiple disks at the same time, but losing one of these disks means losing data; error correction allows for data to be recovered in case of failure but slows down the system as data needs to be read from several places to compute error codes.

Since a RAID has more disks than traditional disk-based storage systems, the probability of disk failures increases. In addition, a failure anywhere in the array can make the entire RAID unusable. In order to improve data integrity, some RAID organizations may reserve one striping unit in each stripe for parity instead of data and uses it to store the XOR of the other striping units. If one disk fails, all its striping units can be recovered with the parity stripes contained in the other disks.

The original specification suggested a number of prototype *RAID levels*, each giving different trade-offs against loss of data, speed and capacity. Over the years, these original configurations have evolved and many more different implementations have appeared, remarkably some that are nested levels. For brevity's sake only the standard levels are described.

- **RAID-0 (striped set without parity):** A RAID-0 stripes data evenly across two or more disks with no parity information, with the number of stripes dictated by the number of disks in the array. This allows stripes to be read off the disks in parallel, thus providing huge bandwidth, but at the cost that the array's entire data may be lost should a failure in any disk happen. It is normally used in order to increase performance in systems where data integrity is not important or to build large virtual disks out of a large number of smaller ones. A RAID-0 can be created with disks of different sizes, but all

disks are treated as if they had the same capacity (the smallest one) and the same performance (the slowest one). It is worth noting that RAID-0 was not one of the original RAID levels but is considered a standard level nevertheless.

- **RAID-1 (mirrored set without parity):** A RAID-1 mirrors a set of data on two or more disks. Applications can benefit from increased read performance if the controller supports split seeks, as disks can be addressed independently. The biggest asset of this configuration, however, is that the reliability of the system augments geometrically as new disks are added to the array. Nevertheless, since all the disks of the array must contain a complete copy of the data, the total capacity of the RAID-1 is reduced to that of the smallest member. This kind of array is useful when read performance or reliability are more important than data storage capacity.

- **RAID-2 (redundancy through Hamming codes):** Data in a RAID-2 configuration is distributed in very small stripes, often at the bit or word level, and error correction is provided by using Hamming codes [57]. The controller synchronizes the disks so that they spin in perfect tandem thus allowing for extremely high data transfer rates. The use of Hamming codes for error correction fixes the number of disks in the array. For example, a `Hamming(7,4)` code—four data bits plus three parity bits—forces the array to use four disks to store data and three for error correction. RAID-2 is the only standard RAID level, other than some implementations of RAID-6, that can accurately detect and correct single-bit corruptions in data. RAID-2 is the only one of the original levels that is not currently used.

- **RAID-3 (striped set with dedicated parity):** RAID-3 uses byte-level striping with a dedicated parity disk. The parity information allows the array to withstand the failure of one disk since the information on the failed disk can be reconstructed by calculating the parity of the remaining disks. RAID-3 configurations cannot serve multiple requests concurrently because, as blocks are spread across all the members of the array, any I/O operation will require activity on every disk. Furthermore, the single parity disk constitutes a bottleneck for writing operations, since any change in a stripe requires updating the parity information. Due to these drawbacks, this configuration is rarely seen in practice.

- **RAID-4 (block level parity):** RAID-4 is identical to RAID-3 but the striping is done at block rather than byte level. Disks can now act independently when only a single block of data is requested, which in turn allows for multiple concurrent requests to be served, if the controller is smart enough to organize disk accesses effectively. Although read requests are highly parallelized with this configuration, writes are penalized since they have to read the parity disk to get the old parity information, compute the new parity and write it back.

- **RAID-5 (striped set with distributed parity):** A RAID-5 configuration uses block level striping like that of a RAID-4, but the parity information is distributed across all the disks instead of only one. The performance impact of this small change is large, since RAID-5 can now support simultaneous write operations. RAID-5 has achieved high popularity due to its low cost of redundancy and performance levels, which are close to those of RAID-1.

- **RAID-6 (striped set with dual distributed parity):** As with RAID-0, this last RAID level was not one of the original levels but over time has grown to be regarded as a standard level. A RAID-6 array is identical to a RAID-5 one, but improves its fault tolerance and recoverability by storing an additional parity block. RAID-6 is the only standard configuration capable of surviving the simultaneous failure of two disks, though at the cost of a relative space inefficiency. However, as an array becomes bigger with more storage devices, this loss in capacity becomes less important, while the probability of two disks failing at once grows.

The use of RAID arrays, however, entails some difficulties. First, the use of parity mechanisms makes small writes expensive. If a write operation is made in whole stripes, it will be fairly easy to compute the new parity for each stripe and write it together with the data. However, if the write is smaller than a stripe, it will be necessary to read the current value of the corresponding parity block and use it to compute the new parity block, in order to keep the stripe's parity consistent. This makes small writes in a RAID about four times slower. Another problem is that all the disks of the array are attached to a single machine, so its memory and I/O system will probably end up being a bottleneck. The most critical problem, however, is the incapability of current RAID mechanisms to effectively use disks of different sizes. Later in this chapter, we will review the research that has been done on the matter (Section 1.5.223).

RAID allows for higher performance (via parallel access), increased reliability (via redundancy) and greater capacity (via aggregation) than using the same disks independently. RAID arrays are used extensively to deploy distributed file systems since they provide high performing, reliable storage in an inexpensive manner, (inexpensive because the file systems do not need to be changed to benefit from the RAID's features). Furthermore, with the appearance of networked RAID protocols [71], RAID configurations have been proved to be valid even in distributed environments, thus extending the technology's usefulness and exceeding the designers' original expectations.

1.4 Object-based Storage

As we have mentioned, as storage infrastructures grow in size and complexity, the amount of data stored on file systems will increase by several orders of magnitude, easily reaching the Petascale and Exascale order. Furthermore, recent industry and academic research trends seem to converge to a shift in storage technology, in which intelligent, self-managed devices will collaborate to provide distributed storage capabilities, effectively carrying part of the burden of the file system. The main problem with this approach, however, is that the current storage interface—blocks—has remained largely unchanged since its introduction.

Traditional (block-based) distributed file systems like NFS [104], AFS [124] or CODA [125] store file data as blocks across the network, decoupling blocks from their association to a file. This separation makes the direct sharing of files between hosts difficult, as a client must have previous knowledge of how the file blocks are laid out over the network in order to acquire them. This usually forces this kind of file systems to provide one or more file servers that are in charge of giving access to the metadata needed to locate the blocks. Of course, centralizing the mapping between files and blocks in file servers introduces bottlenecks and penalizes scalability.

Parallel file systems like OCFS2 [43] follow a different approach: all the machines are organized in a SAN, and hence are able to access any block device. The file system code is then parallelized in order to enable coordinated, concurrent access to the disks. In this manner, every file system client has at the same time the functionality of a file server and is involved in managing the blocks on the storage devices. This can be a disadvantage as the scalability of this approach is limited by the high number of clients managing the block devices explicitly. IBM's GPFS [127] shows

better scaling behavior since it offloads a part of the block management from the clients to Network Storage Device (NSD) servers.

Object-based storage [93, 92] has gradually gained importance as a design paradigm for distributed and parallel file systems, since it seems to provide a feasible solution for these problems. Instead of presenting a storage device as a logical array of unrelated blocks, addressed by a Logical Block Address (LBA), an object-based device appears as a collection of storage objects. A storage object is a logical collection of bytes, with attributes describing its characteristics, high level access methods and security policies. Unlike blocks, an object is of variable size and can be used to store any type of data, such as files or database tables. Objects can be seen as the convergence of files and blocks. Like blocks, objects are a primitive unit of storage that can be accessed directly on the storage device without passing through a server. Like files, objects can be accessed without having to know how they are internally stored in the disk, removing the necessity to access the file system's metadata.

Object-based file systems store the pure file contents—the objects—on an *Object Storage Device* (OSDs), whereas the file namespace and files' metadata are kept in *Metadata Servers* (MDSs). This separation of file content and metadata prevents I/O bottlenecks. File contents can now be spread across a large number of OSDs, while the dedicated MDSs provide fast, independent access to metadata. Once a file system client is properly authorized by the MDS to access a file's objects, all the subsequent I/O operations are served directly by the OSDs, that intelligently manage their own on-disk storage and enforce security policies. In this way read and write operations can be effectively parallelized.

Existing object-based file systems like the Panasas [147] parallel file system, Lustre [18], Ceph [146], and XtreemFS [64, 65] are designed to be deployed in clusters. In this homogeneous environment, all OSDs are equal from a latency and bandwidth point of view with the main cause of failure being hard disks (which is normally handled using RAID). This technology, however, is not yet ready to withstand the heterogeneity inherent to a globally distributed file system although research is currently under way [65, 64].

1.5 Data Distribution, Reliability and Adaptivity

Data reliability and support for scalability as well as the dynamic addition and removal of storage systems is one of the most important issues in designing stor-

age environments. As we have seen, data reliability is commonly achieved by using RAID encoding schemes, which divide data blocks into specially encoded sub-blocks, and place them on different disks to ensure that a certain number of disk failures can be tolerated without losing information [108]. RAID encoding schemes are normally implemented by striping data blocks according to a pre-calculated pattern across all the available storage devices, achieving a nearly optimal performance in small environments. Even though deterministic extensions for the support of heterogeneous disks have been developed [34, 54], adapting the placement to a changing number of disks is cumbersome as a majority of the data may have to be reorganized.

In the following, we review the current data placement strategies that are able to cope with dynamic changes in the capacities or the set of storage devices in the system. These can be classified into two main classes depending on whether the position of a block can be computed in a deterministic manner or not: pseudo-randomized and RAID scaling solutions.

1.5.1 Extensible Pseudo-Randomized Layouts

Pseudo-randomized data distribution is a technique that uses a hash function in order to compute the mapping between blocks and storage devices. Depending on the function, the computation of this mapping can be very fast, and the inherent flexibility of pseudo-randomized hashing makes the strategies based on this principle very adaptable to changes in the storage infrastructure.

Karger et al. proposed an adaptive hashing strategy for homogeneous devices called Consistent Hashing, that is uniform in its data distribution and is competitive when considering adaptivity [70]. This technique uses a hash function h to map each data object to a point in a $[0, 1)$ ring. Once that is done, storage devices are also mapped to the same ring but this time choosing multiple k points in the ring. In order to compute the location of a data block b, the strategy computes the point $p = h(b)$ and follows the ring until it finds the first point where a storage system is mapped. In the resulting distribution, a storage device contains all the blocks located between its point and the previous device point. Thus, the computation of the location of a block takes only an expected number of $O(1)$ steps. Nevertheless, the data structures needed to enforce this strategy require at least $n \log_2 n$ bits to ensure a good data distribution, limiting its applicability.

Brinkmann et al. presented the cut-and-paste strategy as an alternative placement strategy for uniform capacities [23]. Similarly to Consistent Hashing, their

scheme also maps blocks to a $[0, 1)$ interval using a hash function, but the distribution into storage devices is done using the following assimilation method: given n disks, the strategy begins by assigning the interval $[0, 1)$ to the first disk. Then, the algorithm cuts off the range $[1/(n + 1), 1/n]_i$ from every disk i, and concatenates them to create a range $[0, 1/(n + 1)]$ for disk $n + 1$. This scheme requires $O(n \log n)$ bits and $O(\log n)$ steps to compute the position of a block. Furthermore, it keeps the deviation from an ideal distribution extremely small with high probability. Interestingly, the theoretical analysis of this strategy has been experimentally re-evaluated and applied to RAID-0 in Zheng's et al. FastScale [154]. Additionally, Sanders considered the case that disks might fail and suggested to use a set of forwarding hash functions that are reconfigured online taking into account only active disks [123].

Adaptive data placement schemes that are able to cope with arbitrary heterogeneous capacities were introduced by Brinkmann et al. [24]. The presented strategies Share and Sieve are compact, fair, and (amortized) competitive for arbitrary changes from one capacity distribution to another. Share supports heterogeneity by dividing the distribution in two phases. The first phase divides the capacity of each device into several virtual devices, each with δ capacity, which serves to reduce the problem to that of a homogeneous distribution in n devices of capacity δ. Then, the second phase uses a randomized data distribution for uniform capacities, like Consistent Hashing above, to generate the mapping. Since Share has a large space complexity and needs an alternative strategy to work, Brinkmann et al. also introduced Sieve in the same paper, which cuts the $[0, 1)$ ring into $2^{\lceil \log n \rceil + 1}$ equal intervals that are assigned exclusively to storage devices. The strategy then uses L random hash functions $h_1, \dots, h_L$, that are applied in a loop until a successful assignment is found.

Other data placement schemes for heterogeneous capacities are based on geometrical constructions like the *weighted distributed hash tables* [126]: this linear method combines the standard Consistent Hashing approach with a linear weighted distance measure.

All previously mentioned work is only applicable for environments where no replication is required. Certainly, it is easy to come up with proper extensions of the schemes so that no two copies of a block are placed in the same storage device. A simple approach feasible for all randomized strategies to replicate a data block k times is to perform the experiment k times and to remove the selected device after each experiment. Nevertheless, it has been shown that fairness cannot be guaranteed for these simple strategies and that some capacity will be wasted [21].

The algorithm proposed in SCADDAR [50] moves a data block only if the destination disk is one of the newly added disks. This approach reduces data migration significantly, but produces an unbalanced data distribution after subsequent upgrade operations.

The first methods with dedicated support for replication were proposed by Honicky and Miller [60, 61]. RUSH (Replication Under Scalable Hashing) maps replicated objects to a scalable collection of storage servers according to user-specified server weighting. When the number of servers changes, RUSH tries to redistribute as few objects as possible to restore a balanced data distribution while ensuring that no two replicas of an object are ever placed on the same server. Depending on the technique used to map blocks to the appropriate servers the strategy can be specialized in three variants: $RUSH_P$, $RUSH_T$, and $RUSH_R$. We will review each of them in detail in Chapter 2_{29}.

A drawback of the RUSH-variants is that they require that new capacity is added in chunks, where each chunk is based on servers of the same type and the number of disks inside a chunk has to be sufficient to store a complete redundancy group without violating fairness and redundancy. The use of sub-clusters is required to overcome the problem if more than a single block of a redundancy group is accidentally mapped to the same hash-value. This property leads to restrictions for bigger numbers of sub-blocks. In this case, prime numbers can be used to guarantee a unique mapping between blocks of a redundancy group and the servers inside a chunk.

CRUSH is derived from RUSH, and supports different hierarchy levels that provide the administrator finer control over the data placement in the storage environment [145]. The algorithm accommodates a wide variety of data replication and reliability mechanisms and distributes data in terms of user-defined policies.

Amazon's Dynamo [38] uses a variant of Consistent Hashing with support for replication where each node is assigned multiple "tokens" (positions in a $[0-1) \in \mathbb{R}$ ring) chosen at random that are used to partition the hash space. The number of tokens that a node is responsible for is decided based on its capacity, thus taking into account the heterogeneity in the performance of nodes. In addition, Dynamo uses a membership model where each node is aware of the data hosted by its peer nodes, and requires that each node actively gossips the full routing table. Dynamo's authors, however, claim that scaling this design to run with tens of thousands of nodes is not easy because the complexity of the routing table increases with the size of the system, which may make the strategy unsuitable for Exascale storage in its current form.

Brinkmann et al. showed that a huge class of placement strategies cannot preserve fairness and redundancy at the same time and presented a placement strategy called Redundant Share for an arbitrary fixed number k of copies for each data block, which is able to run in $O(k)$. The strategy shows a competitiveness[3] of $\log n$ for the number of replacements in case of a change of the infrastructure [21], but can be reduced to $O(1)$ by breaking the heterogeneity of the storage systems [20]. The Spread strategy is also worth mentioning, having similar properties to those of Redundant Share [91].

[3] *A strategy is ϵ-competitive if it can store at least $(1-\epsilon)$-times the data stored by an optimal strategy.*

1.5.2 Extensible RAID Layouts

There are several approaches to improve the extensibility of RAID systems. The reshape toolkit in the Linux MD driver implementing RAID-5 uses a fixed-size window to write mapping metadata. The main problem with this approach, however, is that user requests to this window must wait until all its data blocks have been migrated, which is inefficient.

Gonzalez and Cortes proposed the Gradual Assimilation algorithm [52] to control the overhead of upgrading a RAID-5 system, which still has a large redistribution cost since all parities still need to be modified after a data migration.

The US Patent #6,000,010 *"Method of increasing the storage capacity of a level five RAID disk array by adding, in a single step, a new parity block and N–1 new data blocks which respectively reside in a new columns, where N is at least two"* presents a method to upgrade RAID-5 volumes called MDM. MDM reduces data movement by exchanging some data blocks between the original disks and the new disks. It also eliminates parity modification costs since all parity blocks are maintained, but it is unable to increase (only maintain) the storage efficiency by adding new disks [76].

More recently, FastScale [154] attempted to minimize data migration by moving only data blocks between old and new disks. It also optimizes the migration process by accessing physically sequential data with a single I/O request and by minimizing the number of metadata writes. At the moment, however, it cannot be used with RAID-5, only RAID-0, which limits its applicability.

GSR [151] divides data on the original array into two consecutive sections. During RAID-5 upgrades, GSR moves the second section of data onto the new disks keeping the layout of most stripes, thus minimizing data migration and parity updates. Its main limitation is the performance of the array after upgrades, since accesses to the first section are served by original disks only, and accesses to the second section are served by new disks only.

1.6 Hierarchical Storage and Layout Optimization

There has been extensive research on hierarchical architectures and data layout optimizations, in order to improve the performance, reliability and scalability of storage media. Since Chapter 5_{107} proposes a solution that mixes both concepts, it is interesting to have a look at the existing literature first. Note that neither of the works discussed herein tackle the problem of upgrading a RAID array.

[4] *NVRAM*

The first publication on hierarchical storage that we know of is a 1992 study by Baker et al. [12] that considered the use of battery-backed non-volatile RAM[4] in distributed file system servers to reduce write traffic. Later, HP's AutoRAID [148] extended traditional RAID organizations by partitioning storage in a mirrored zone and a RAID-5 zone. Writes are initially made to the mirrored storage and later migrated in large chunks to the RAID-5 zone, thus reducing the space overhead of redundancy data and increasing parallel bandwidth for subsequent reads.

Hu et al. [152, 103] proposed an architecture called Disk Caching Disk (DCD), where an additional HDD is successfully used as a cache in order to convert small random writes into large log appends, thus demonstrating that a dedicated cache can improve the overall I/O performance. Similarly to DCD, iCache [59] added a log-disk along with a piece of NVRAM to create a two-level hierarchy cache for iSCSI requests, coalescing small requests into large ones before writing data and also improving performance.

A study by Uysal et al. [137] offered a cost-performance analysis of replacing some or all of the disks with microelectromechanical storage (MEMS) in the storage hierarchy. It showed that for those workloads where performance is more important than capacity, MEMS can be competitive.

More recently, several HDD+SSD hybrid architectures have been proposed in order to combine the benefits of both magnetic media and solid state devices. For instance, Intel®'s Turbo Memory [87] and Windows ReadyBoost [94] used flash-based storage as a cache on top of hard drives to improve performance. In contrast, Griffin [132] used the HDD as a write cache to extend the lifetime of SSDs.

Researchers have also considered placing SSDs and HDDs at the same level in the storage hierarchy. Combo Drive [111] concatenated sectors from an SSD and an HDD to create a continuous address range, where data was allocated according to certain heuristics. Similarly, Koltsidas et al. divided a database store between the two media types based on several on-line algorithms and successfully improved its performance [74].

Regarding data layout optimizations, early works by Wong [149], Vongsathorn et al. [140] and Ruemmler and Wilkes [120] argued that placing frequently accessed data in the center of the disk served to minimize the expected head movement. Specifically, Ruemmler and Wilkes demonstrated that the best results in I/O performance came from relatively infrequent shuffling (once a week) with small granularity (block or track size). Akyurek and Salem also proved the importance of data reorganization at the block level, as well as the advantages of copying over shuffling [5].

In 2004, Li et al. propose C-Miner [81], which uses data mining techniques to model the correlations between different block I/O requests. They show that correlation-directed prefetching and data layout can reduce average I/O response time by 12–25%. ALIS [62] and, more recently, BORG [16], reorganize frequently accessed blocks (and block sequences) so that they are placed sequentially on a dedicated area on the disk, thus improving the overall performance of the storage device. Neither of them explores multi-disk systems, however.

1.7 Characterization of Storage Behavior

In Chapter 4 [83], we develop an extensive study on the long-term behavior of stored data in an attempt to determine common optimizable access patterns. Nevertheless, effectively characterizing file system behavior is difficult because of several problems. First, there is a wide range of workloads to take on, each with its particular type of access patterns, data and intended users. Second, there are technical difficulties associated with the generation of traces, such as the performance impact incurred by enabling the tracing framework or the large amounts of data generated by it. Third, usage semantics of file systems change as time goes on which can render current models invalid for future workloads. In spite of these problems, several studies have been conducted over time that have provided a global view on common file system behaviors. Let us review some of them.

Early trace-based file system studies like those from Smith [131] and Ousterhout et al. [105] provide useful observations on file system behavior that, though useful, have been slowly losing relevance due to changes in storage semantics. Smith studied text-based user files for thirteen months which are very different from the large multimedia files of current user workloads. Ousterhoust's analysis traced three servers running BSD over a period of three to four days which is insufficient to predict long-term access trends.

Ramakrishnan et al. were the first to examine traces collected from commercial customer sites over several environments. They observed that only a relatively small percentage of all data was active at a time and that it received a considerable amount of accesses. They also described that a large portion of active data (23%–37%) was shared by multiple processes over time [115]. Nevertheless, their analysis focused on file accesses with a resolution of hundreds of nanoseconds and their results may not apply to access patterns seen over longer spans of time.

In 1996, Gibson, Miller and Long [49] conducted a long-term study on file system activity on different UNIX environments. In particular, they were interested in long-term behavior of files over a distributed file system to find common activity trends and access patterns. Consistently with previous works, they described that 90% of all files were not used after creation, and that approximately 1% of all files were used daily. Furthermore, they determined that files were usually short-lived and, that if they were not used immediately after being created, they would never be. However, their study is file-based which limits its applicability when considering block access behavior.

In an attempt to track how the behavior of a file system changes over time, Roselli et al. [118] measured a wide range of file systems and compared their results with those from the Sprite study, conducted almost a decade earlier [13]. They noted that I/O load varied greatly depending on the environment studied, but that file access patterns were bimodal in all environments: files were mostly read or mostly written. Most interestingly, they described that block lifetimes had increased since past studies, as well as maximum file sizes. Their primary interest was determining how caching, memory mapping and file system parameters affect disk behavior.

Ellard et al. [42] analyzed NFS traffic for research and email environments and found that blocks died quickly in both, and that many read access patterns classified as "random" by NFS servers were in fact long reads composed of sequential sub-runs. Though they focused on determining access patterns for individual files and blocks, they did not consider block or file sharing semantics.

In 2008, Leung et al. examined CIFS traffic for two enterprise servers during three months [78]. They described that read-write and random access patterns were more common than previously thought, that file sharing by clients was rarely concurrent and that a small fraction of clients accounted for most file activity. Again, this study used files as its basis and even though it analyzed file sharing by multiple clients, it only considered concurrent sharing, disregarding temporal sharing.

In the most recent study that we are aware of (2011), Chen et al. [31] analyzes data access patterns in two CIFS file system traces from a production enterprise datacenter. Among other relevant observations, they note that client access patterns reflect the aggregate behavior of the system's human users, with session activity corresponding exactly to the U.S. work day/week (including the rushes to meet daily/weekly deadlines). They also observe that application generated I/O departs significantly from human patterns. This study is particularly interesting as it contributes several observations and implications for both clients and servers.

These studies show that data access patterns are as variable as the semantics of the storage system contents, and that it is as important to determine what is going to be stored as what its prospective clients are meant to do with it.

Chapter 2

Scalable Data Distribution

"We are just an advanced breed of monkeys on a minor planet of a very average start. But we can understand the Universe. That makes us something very special."
— Stephen W. Hawking

2.1 Motivation

As we mentioned in Chapter 0$_{1}$, the ever-growing creation of, and demand for, massive amounts of data requires highly scalable storage solutions. The most flexible approach is to use a pool of storage devices that can be expanded and downgraded as needed by adding new storage devices or removing older ones. This approach, however, necessitates a scalable solution for locating data items in such a dynamic environment.

An ideal data distribution solution should be *fair* and distribute data blocks taking into account the capacity of each device. It should also be *adaptable* and require minimal data migration when the number of devices in the system changes. These challenges have led to the proposal of several families of data distribution strategies (see Table 2.1$_{31}$).

Table-based strategies can provide an optimal mapping between data blocks and storage devices, but obviously do not scale to large systems because the tables used grow linearly in the number of data blocks.

Rule-based methods, on the other hand, tend to run into fragmentation problems, which require a periodic defragmentation in order to preserve scalability. Furthermore, they force the migration of exceedingly large amounts of data when storage devices are added or removed [52, 53].

Hashing-based strategies use a compact function h in order to map *balls* with unique identifiers out of some large universe U into a set of *bins* called S so that the balls are evenly distributed among the bins. In our case, balls are data items and bins are storage devices. Given a static set of devices, it is possible to construct a hash function so that every device gets a fair share of the data load.

Standard hashing techniques, however, do not adapt well to a changing set of devices: consider, for example, the hash function $h(x) = (a \cdot x + b) \bmod n$, where S represents the set of storage devices and is defined as $S = \{0, \ldots, n-1\}$. If a new device is added, we are left with two choices: either replace n by $n+1$, which would require virtually all the data to be relocated; or add additional rules to $h(x)$ to force a certain set of data blocks to be relocated on the new device in order to get back to a fair distribution, which, in the long run, destroys the compactness of the hashing scheme.

Pseudo-randomized hashing schemes that can adapt to a changing set of devices have been proposed and theoretically analyzed, showing promising results. The most popular is probably Consistent Hashing [70], which is able to evenly distribute single copies of each data block among a set of storage devices and to adapt to a changing number of disks. We will show that these pure randomized data distribution strategies have, despite their theoretical perfectness, serious drawbacks when used in very large systems. Especially, many of their beneficial properties are only achieved *if a certain level of randomness can be achieved*. This is particularly important because, as we will show in Chapter 365, it has a serious influence either on the necessary amount of memory required by a strategy, or on the performance it delivers, which may render it infeasible in large-scale environments.

Besides adaptivity and fairness, *redundancy* is important as well. Storing just a single copy of a data item in real systems is dangerous because if a storage device fails, all of the blocks stored in it are lost. It has been shown that simple extensions of standard randomized data distribution strategies to store more than a single data copy are not always capacity efficient [21].

Nevertheless, despite the sheer amount of randomized data distribution strategies that have been proposed over the years, there does not exist a formal analysis that evaluates them in a common environment. We believe it is a good idea to explore the advantages and disadvantages of current data distribution strategies, as

Table 2.1: Pros and cons of several approaches to data distribution

Strategy	Advantages	Limitations
Table-based $T[b] \mapsto D$	Extremely fast† lookup Easily adaptable to new devices	Poor scalability (tables grow linearly with number of blocks) Computationally intensive reversibility for large table sizes
Rule-based $f(b) \mapsto D$	Fast† lookup Easily reversible‡	May run into fragmentation problems Adapting to new devices involves huge data migrations
Pseudo-randomized $h_n(b) \mapsto D$	Moderately* fast† lookup Inherently fair Extremely flexible (naturally adapts to new devices)	May require substantial memory to provide desired randomness Not easily reversible‡

Definitions used: b, data block ID; D, device ID; $T[x]$, content of table T for block ID x; $f(x)$, result of applying rule f to block ID x; $h_n(x)$, value of applying one, several or a combination of n hash functions $h_0, \ldots, h_n$ to block ID x.

† Depending on particular implementation.

‡ It is possible to compute the collection of blocks managed by a particular device.

* Depending on the number of hash functions applied.

dissecting their strengths and limitations can be useful for storage researchers and developers alike, and may lead to the invention of better distribution mechanisms.

A second motivation is to use this information to design a large-scale data distribution strategy that shares the advantages of current solutions and shows none of their limitations. Based on these motivations, the main contributions of this chapter can be summarized in the two following points:

1. The contribution of the first comparative evaluation of different hashing-based distribution strategies that are able to replicate data in a heterogeneous and dynamic environment. This comparison shows the strengths and drawbacks of the different strategies as well as their constraints. Such comparison is novel because hashing-based data distribution strategies have been mostly analytically discussed, with only a few implementations available, and in the context of peer-to-peer networks with limited concern for the fairness of the data distribution [135]. Only a few of these strategies have been implemented in storage systems, where limited fairness immediately leads to a strong increase in costs [22, 146].

2. The introduction of a novel distribution mechanism called Random Slicing, which overcomes the drawbacks of current randomized data distribution strategies by incorporating lessons learned from table-based, rule-based and pseudo-randomized hashing strategies. Random Slicing keeps a small table with information on previous storage system insertions and removals, which helps to contain the required amount of randomness that needs to be computed and thus reduces the amount of necessary main memory by several orders of magnitude.

It is important to note that all randomized strategies map (virtual) addresses to a set of disks, but do not define the placement of the corresponding block on the disk surface. This placement on the block devices has to be resolved by additional software running on the disk itself. Therefore, we will assume for the remainder of the chapter that the presented strategies work in an environment that uses object-based storage. Unlike conventional block-based hard drives, object-based storage devices (OSDs) manage disk block allocation internally, exposing an interface that allows others to read and write to variably-sized, arbitrarily-named objects [10, 39].

2.2 Research Model

Our research is based on an extension of the standard "balls into bins" mathematical model [69, 100, 114]. Let $\{0, \ldots, M-1\}$ be the set of all identifiers for the balls and $\{0, \ldots, N-1\}$ be the set of all identifiers for the bins, where each ball represents a data block and each bin a storage device. Suppose that the current number of balls in the system is $m \leq M$ and that the current number of bins in the system is $n \leq N$. We will often assume for simplicity that the balls and bins are numbered in a consecutive way starting with 0, but any numbering that gives unique numbers to each ball and bin would work for our strategies.

Suppose that bin i can store up to b_i (copies of) balls. Then we define its relative capacity as $c_i = b_i / \sum_{j=0}^{n-1} b_j$. We require that, for every ball, k copies must be stored in different bins for some fixed k. In this case, a trivial upper bound for the number of balls the system can store while preserving fairness and redundancy is $\sum_{j=0}^{n-1} b_j / k$, but it can be much less than that in certain cases. We term the k copies of a ball a *redundancy group*.

Placement schemes for storing redundant information can be compared based on the following criteria (see also the work by Brinkmann et al. [24]):

- *Capacity Efficiency and Fairness.* A scheme is called *capacity efficient* if it allows us to store a near-maximum number of data blocks. We will see in the following that the fairness property is closely related to capacity efficiency, where *fairness* describes the property that the number of balls *and* requests received by a bin are proportional to its capacity. Also, we will often refer to the *competitiveness* of a replication scheme: a scheme is called ϵ-competitive if it is able to store at least $(1 - \epsilon)$-times the amount of data that could be stored by an optimal strategy.

- *Time Efficiency.* A scheme is called *time efficient* if it allows a fast computation of the position of any copy of a data block without the need to refer to centralized tables. Schemes often use smaller tables that are distributed to each node that must locate blocks.

- *Compactness.* We call a scheme *compact* if the amount of information the scheme requires to compute the position of any copy of a data block is small (in particular, it should only depend on n—the number of bins).

- *Adaptivity.* We call a scheme *adaptive* if it only redistributes a near-minimum amount of copies when new storage is added in order to get back into a state of fairness. Therefore, in Section 2.751 we compare the different strategies with the minimum amount of movements which is required to keep the fairness property.

The better a distribution strategy performs under all of these criteria, the closer it is to being an ideal solution.

2.3 Randomized Data Distribution

For the reader's convenience, we present in this section a short description of the applied data distribution strategies we will evaluate. We start with Consistent Hashing and Share, which can, in their original form, only be applied for $k = 1$ (i.e. only one copy of each data block) and therefore lack support for redundancy. Both strategies are used as sub-strategies inside some of the investigated data distribution strategies. Besides their usage as sub-strategies, we will also present a simple replication strategy which can be based on any of these simple strategies. Afterwards, we present Redundant Share and RUSH, which directly support data replication.

2.3.1 Consistent Hashing

We start with the description of the Consistent Hashing strategy, which solves the problem of (re-)distributing data items in homogeneous systems [70]. In Consistent Hashing, both data blocks and storage devices are hashed to random points in a $[0, 1)$-interval, and the storage device closest to a data block in this space is responsible for that data block. In order to produce a balanced distribution, however, storage devices need to be associated to more than one point in the ring and, therefore, the strategy needs to produce $k \cdot \log n$ points for each device for some constant k. As we will see later, this requirement can affect the applicability of the strategy.

Consistent Hashing ensures that adding or removing a storage device only requires a near minimal amount of data replacements to get back to an even distribution of the load, because the only blocks that migrate are those closest to one of the new device's associated points and no blocks can move between old devices. However, this technique cannot be applied well if the storage devices can have arbitrary non-uniform capacities since in this case the load distribution has to be adapted to the capacity distribution of the devices. The memory consumption of Consistent Hashing heavily depends on the required fairness. Using only a single point for each storage devices leads to a load deviation of $n \cdot \log n$ between the least and heaviest loaded storage devices. Instead it is necessary to use $\log n$ virtual devices to simulate each physical device, and to respectively throw $\log n$ points for each device to achieve a constant load deviation.

2.3.2 Share

The Share strategy supports heterogeneous environments by introducing a two stage process [24]. In the first stage, the strategy randomly maps one interval for each storage system to the $[0, 1)$-range. The length of these intervals is proportional to the size of the corresponding storage systems (plus some stretch factor σ) and can cover the $[0, 1)$-ring many times. In addition, the interval for each device is divided into equally-sized ranges of capacity δ, called *virtual bins*. The purpose of this division is to simplify the initial problem to that of data distribution with uniform capacities, where the distribution targets are equally-sized bins. Share now only needs to use an adaptive strategy for homogeneous storage systems, like Consistent Hashing, to get the storage system for which the corresponding interval includes this point.

The analysis of the Share strategy shows that it is sufficient to have a stretch factor $\sigma = O(\log N)$ to ensure correct functioning and that Share can be implemented in expected time $O(1)$ using a space of $O(\sigma \cdot k \cdot (n + 1/\delta))$ words (without considering the hash functions), where δ characterizes the required fairness. Share has an amortized competitive ratio of at most $1 + \epsilon$ for any $\epsilon > 0$. Nevertheless, we will show that, similar to Consistent Hashing, the memory consumption heavily depends on the expected fairness.

2.3.3 Trivial Data Replication

Consistent Hashing and Share are, in their original implementations, unable to support data replication or erasure codes, since it is always possible that multiple stripes belonging to the same stripe set are mapped to the same storage system and that data recovery in case of failures becomes impossible. Nevertheless, it is easy to imagine strategies to overcome this drawback and to support replication strategies by, e.g., simply removing all previously selected storage systems for the next random experiment for a stripe set. Another approach, used inside the experiments in this chapter, is to simply perform as many experiments as are necessary to get enough independent storage systems for the stripe set. It has been shown that this trivial approach wastes some capacity [21], but we will show that this amount can often be neglected.

2.3.4 Redundant Share

Redundant Share [21] was developed to support the replication of data in heterogeneous environments. The strategy orders the bins according to their weights c_i and sequentially iterates over them. The basic idea is that the weights are calculated in a way that ensures perfect fairness for the first copy and to use a recursive descent to select additional copies. Therefore, note that the strategy needs $O(n)$ rounds for each selection process, where n is the number of storage devices.

Each round starts by selecting a random value $v \in [0, 1)$, which is calculated based on the data block identifier and the identifier of the bin that is being tested to hold the data block. For each bin i, if v is smaller than the adapted weight $\check{c}_i = 2 \cdot c_i / \sum_{j=i}^{n-1} b_j$ of bin i—which represents the probability that bin i can hold the ball while maintaining a uniform distribution—the bin is chosen for the primary copy. The remaining copies are placed by repeating the process with all bins with a smaller weight.

This algorithm is $\log n$-competitive concerning the number of replacements if storage systems enter or leave the system. The authors of the original strategy also presented extensions of Redundant Share, which are $O(1)$-competitive concerning the number of replacements as well as strategies which have $O(k)$ runtime [20]. Both strategies rely on Share and we will discuss in the evaluation section why they are not feasible in realistic settings.

2.3.5 RUSH

RUSH algorithms are based in the notion that, as large storage systems expand, new capacity is typically added several disks at a time. Thus, the RUSH algorithms all proceed in two stages, first identifying the appropriate sub-cluster in which to place an object, and then identifying the disk within that particular sub-cluster [60, 61]. Within a sub-cluster, replicas assigned to the sub-cluster are mapped to disks using a hash function and prime number arithmetic that guarantees that no two replicas of a single object can be mapped to the same disk.

The selection of sub-clusters is a bit more complex and differs between the three RUSH variants: RUSH_P, RUSH_R, and RUSH_T. RUSH_P considers sub-clusters in the reverse of the order they were added, comparing the hash value of the object with the ratio of the weight of the most recently added sub-cluster to the total weight of the system. If the hash value is smaller than this ratio, the search terminates and the object is placed in that sub-cluster. Otherwise, RUSH_P discards the most recently added sub-cluster and repeats the process with the remaining ones. RUSH_R works in a similar way, but it determines the number of objects in each sub-cluster simultaneously, rather than requiring a draw for each object. RUSH_R uses the same ratio as RUSH_P in order to determine the placement of objects, but these values are used as parameters to a draw from the hypergeometric[1] distribution. The result of this operation yields the number of replicas that belong in the most recently added sub-cluster.

RUSH_T improves the scalability of the system by descending a tree to assign objects to sub-clusters; this reduces computation time to $\log c$, where c is the number of sub-clusters added. RUSH_T is similar to RUSH_P, except that it uses a binary tree data structure rather than a list, where each tree node is aware of the weights of its child sub-trees. Also, each tree node has a unique identifier which is used as a parameter to the hash function. In order to locate the appropriate sub-cluster, the algorithm computes the hash value v of the object *and* the current tree node, beginning at the root of the tree. Then, v is compared with the weight of the sub-

[1] *The hypergeometric distribution applies to sampling without replacement from a finite population whose elements can be classified into two mutually exclusive categories like Success/Error. As random selections are made from the population, each subsequent draw decreases the population causing the probability of success to change with each draw.*

trees, and the algorithm chooses the sub-tree with fewer weight. This process is repeated until a leaf node is reached.

The main drawback of the RUSH-variants is that they require that new capacity is added in chunks, where each chunk is based on servers of the same type and the number of disks inside a chunk has to be sufficient to store a complete redundancy group without violating fairness and redundancy, which leads to restrictions for larger numbers of sub-clusters.

2.4 Proposal: Random Slicing

In this section we describe our proposal for a new data distribution strategy called Random Slicing. This strategy tries to overcome the drawbacks of current randomized data distribution strategies by incorporating lessons learned from table-based and pseudo-randomized hashing strategies. In particular, it tries to reduce the required amount of randomness necessary to keep a uniform distribution, which can cause memory consumption problems.

2.4.1 Description

Random Slicing is designed to be fair and efficient both in homogeneous and heterogeneous environments and to adapt gracefully to changes in the number of bins. Suppose that we have a random function $h : \{1, \dots, M\} \rightarrow [0, 1)$ that maps balls uniformly at random to real numbers in the interval $[0, 1)$. Also, suppose that the relative capacities for the n given bins are $(c_0, \dots, c_{n-1}) \in [0, 1)^n$ and that it is always the case that $\sum_{i=0}^{n-1} c_i = 1$.

The strategy works by dividing the $[0, 1)$ range into intervals and assigning them to the bins currently in the system (see Figure 2.138). Notice that the intervals created do not overlap and completely cover the $[0, 1)$ range. Also note that bin i can be responsible for several non-contiguous intervals $P_i = (I_0, \dots, I_k)$, where $k < n$, which will form the *partition* of that bin. To ensure fairness, Random Slicing will always enforce that the accumulated capacity of all intervals for bin i equals its relative capacity, that is $\sum_{j=0}^{k-1} |I_j| = c_i$.

In an initial phase, i.e. when the first set of bins enters the system, each bin i is given only one interval of length c_i, since this suffices to maintain fairness. Whenever new bins enter the system, however, relative capacities for old bins change due to the increased overall capacity. To maintain fairness, Random Slicing shrinks

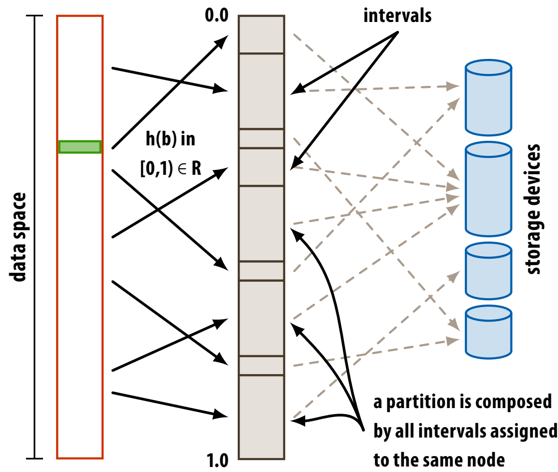


Figure 2.1: Overview of Random Slicing's data distribution. The data space is divided into numeric intervals that are assigned to storage devices. A storage device can be responsible for several intervals that configure the device's partition of the data space. The size of each partition is computed so that the amount of data contained in it matches the capacity of the device relative to the overall storage capacity. Blocks are assigned to intervals using a pseudo-randomized hash function that guarantees uniformity of distribution.

existing partitions by splitting the intervals that compose them until their new relative capacities are reached. Splitted interval fragments can now be used to create partitions for the new bins that maintain the fairness constraint.

The splitting mechanism works as follows. First, the strategy computes how much partitions should be reduced by in order to keep the fairness of the distribution. Since the global capacity has increased, each partition P_i must be reduced by $r_i = c_i - \check{c}_i$, where $\check{c}_i$ corresponds to the new relative capacity of bin i.

Partitions become smaller by releasing or splitting some of their intervals, thus generating *gaps*, which can be used for new intervals. This way, the lengths of partitions for the old bins already represent their corresponding relative capacities and it is only necessary to use these gaps to create new partitions for the newly added bins.

The time efficiency and the memory consumption of the strategy, of course, crucially depend on the maximum number of intervals being managed. In the following theorems we derive a theoretical upper bound for this number.

Theorem 2.4.1. *Assume an environment where storage systems can only be added. In this case, adding n storage systems leads to at most* $(1/2) \cdot n \cdot (n+1)$ *intervals.*

Proof. We use an induction technique to prove the theorem. For the base case it holds that adding the first storage system leads to 1 interval and it holds that $(1/2) \cdot 1 \cdot 2 = 1$. In the following we show the inductive step from change n to

change $(n+1)$. We show that adding a new storage system leads to at most n new intervals and that therefore the number of intervals after the insertion is at most:

$$\begin{aligned}(1/2)\cdot n\cdot(n+1)+n &= (1/2)\cdot(n^2+n)+n\\ &= (1/2)\cdot(n^2+n+2n)\\ &= (1/2)\cdot(n^2+3n) \qquad (2.1)\\ &\le (1/2)\cdot(n^2+3n+2)\\ &= (1/2)\cdot(n+1)\cdot(n+2),\end{aligned}$$

which is the inductive step.

The number of intervals is smaller than n before a new storage system is added. Then, a part of the intervals of each storage system already within the system has to be assigned to the new storage system. Assume now for each existing storage system that we pick an arbitrary interval first. If the size of this interval is smaller than the interval length, which has to be assigned to the new storage system, we assign the complete interval to the new storage system. The number of intervals does not change in this case and we continue with the next intervals until we reach one interval which cannot be completely assigned to the new storage system. In this case, we split this interval and assign one part to the new storage system and keep the other for the previously existing one. Therefore, the number of intervals increases by at most one for each previously existing storage system (or stays constant if the last assigned interval will be completely assigned to the new storage system). □

In the following, we investigate an environment where storage systems can also be removed. We assume that the number of storage systems is not decreasing for a long time in a typical storage environment and that new storage systems are always bigger than storage systems which have been removed. We suppose a situation where the number of storage devices added to the environment steady state, is at most equal to the number of devices previously removed.

Theorem 2.4.2. *Assume that there have been at most n storage systems in the environment at any time. Then the number of intervals will necessarily be smaller than* $(1/2)\cdot n\cdot(n+1)$ *in the steady state.*

Proof. Assume that k storage systems have been removed and no new storage systems have been added. In this case it can occur that the number of intervals becomes bigger than $(1/2)\cdot n\cdot(n+1)$. Now assume that at least k new storage systems have been added to the environment and the system is back in a steady state. We

will show in this case that the number of intervals is at most $(1/2) \cdot n \cdot (n+1)$, where n is the number of all storage systems which have previously been part of the environment. We use a technique inspired by the zero-height-strategy proposed by Brinkmann et al. [23], and assign first all intervals which have been previously assigned to the removed storage system, to the new storage systems. This is possible as the new storage systems are at least as big as the removed storage systems. If the new storage systems are bigger than the removed ones, additional intervals lengths have to be assigned to them. In this case, we just assign the remaining capacity in the same way as we did in Theorem 2.4.138. □

As we have seen, the theoretical upper bound on the total number of intervals is bounded by $O(n^2)$, which means that the pressure on performance and memory consumption could become a challenge after several reorganizations. Notice, however, that the theorem assumes a worst-case scenario where storage systems are added one by one and that, as we will demonstrate with the experiments of Section 2.644, an appropriate algorithm can significantly reduce the number of intervals to a manageable amount if several storage systems are added in bulk.

2.4.2 Interval Creation Algorithm

The goal of the interval creation algorithm is to split existing intervals in a way that allows new intervals to be created while maintaining the relative capacities of the system. Note that the strategy's memory consumption directly depends on the number of intervals used and, therefore, the number of new intervals created in each addition phase can hamper scalability. We briefly explain two interval creation strategies and two variants.

- **Greedy:** This algorithm tries to collect as many complete intervals as possible and will only split an existing interval as a last resort. Furthermore, when splitting an interval is the only option, the algorithm tries to expand any adjacent gap instead of creating a new one. Once enough gaps are collected to produce an even distribution, they are assigned sequentially to new partitions.

- **CutShift:** This algorithm also tries to collect as many complete intervals as possible, but, when it is necessary to split an interval, alternates between splitting it by the beginning or by the end in an attempt to maximize gap

length. Once enough gaps are collected to produce an even distribution, they are assigned sequentially to new partitions.

- **Greedy+Sorted:** A variant of Greedy where the largest partitions are assigned greedily to the largest gaps available.
- **CutShift+Sorted:** A variant of CutShift where the largest partitions are assigned greedily to the largest gaps available.

Note that these strategies are intentionally simple since our intention is that interval reorganizations can be computed as fast as possible in order to reduce the reconfiguration time of the storage system.

An example of the CutShift+Sorted reorganization can be seen in Figure 2.242, where two new bins B_3 and B_4, representing a 33% capacity increase, are added to bins B_0, B_1, and B_2. Figure 2.2a42 shows the initial configuration and the relative capacities for the initial bins. Figure 2.2b42 shows that the partition of B_0 must be reduced by 0.06, the partition of B_1 by 0.11, and the one of B_2 by 0.16, whereas two new partitions with a size of 0.14 and 0.19 must be created for B_3 and B_4. The interval $[0.1, 0.2) \in B_1$ can be completely cannibalized, whereas the intervals $[0.0, 0.1) \in B_0$, $[0.2, 0.6) \in B_2$, and $[0.7, 0.9) \in B_1$ are split while trying to maximize gap lengths. Figure 2.2c42 shows that the partition for B_3 is composed of intervals $[0.23, 0.36)$ and $[0.7, 0.71)$, while the partition for B_4 consists only of interval $[0.04, 0.23)$.

2.4.3 Data Lookup

Once all partitions are created, the location of a data item/ball b can be easily determined by calculating $x = h(b)$ and retrieving the bin associated with it. Notice that some balls will change partition after the reorganization, but as partitions always match their ideal capacity, only a near minimal amount of balls will need to be reallocated. Furthermore, if $h(b)$ is uniform enough and the number of balls in the system is significantly larger than the number of intervals (both conditions easily feasible), the fairness of the strategy is guaranteed.

2.4.4 Fault Tolerance

As explained in Section 2.3.335, Random Slicing provides fault tolerance to data loss by using data replication. In order to ensure data resilience, and given a block

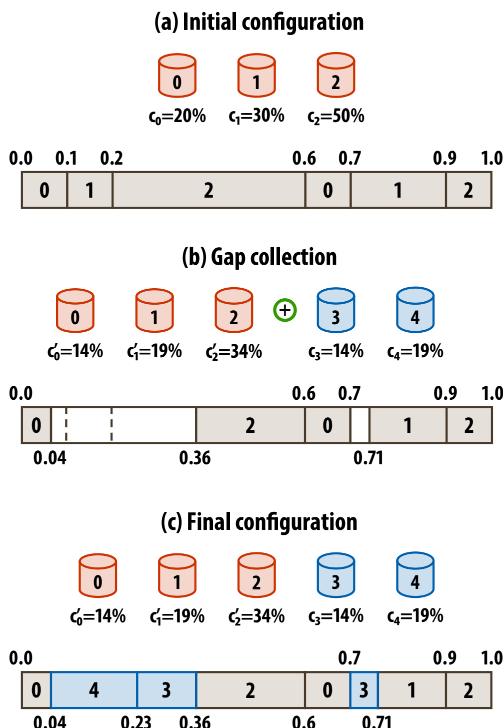


Figure 2.2: Example of the CutShift+Sorted interval management algorithm. Two new devices B_3 and B_4 are added to an existing system, each representing an increase in capacity of 14% and 19%, respectively. The algorithm first computes the new relative capacities (c_i) of the existing partitions and proceeds to create *gaps* by either assimilating or slicing intervals. Interval [0.1, 0.2) is completely assimilated, while intervals [0.0,0.1) and [0.2, 0.6) are cut. Note how the algorithm shifts the cutting point to the beginning or the end of the interval in an attempt to increase the potential length of the current gap. When all gaps are created, the largest partitions are assigned to the largest intervals in an greedy fashion, in order to avoid unnecessary splitting the new partitions.

identifier, the strategy simply needs to perform as many random experiments as needed to get enough independent partitions to place each replica. In the worst case, this trivial approach can have a negative effect in performance (due to a potentially high number of random experiments) and fairness (due to an unbalanced placement of copies). Nevertheless, as we will see in Section 2.7$_{51}$, the practical impact in our strategy is neglectable.

Concerning metadata, the interval table should be safe-guarded at all costs, since it is the key element that allows clients to locate data blocks. Nevertheless, for performance reasons it should be kept in memory in order to be able to locate the appropriate partitions efficiently, which could cause problems in case of unexpected shutdowns or node failures. In case of metadata loss, the strategy could recover in three ways:

1. A client node could rebuild the interval table simply by replaying the successive changes in devices (and capacity) made to the storage system. This

data could be kept in an external log, or could be provided by the system administrator if needed.

2. As we will see in Section 2.6_{44}, the memory footprint of the interval table is small even for tens of thousands of storage devices. Since the interval table is only modified when changes are made to the storage system, it would be reasonable to keep a backup copy of the table in secondary storage.

3. All client nodes must have an in-memory copy of the interval table in order to access the storage system; it would be easy to transfer this information from an active client to a newly-recovered one.

2.4.5 Extensions

Note that up to now we have discussed Random Slicing's capability to distribute data blocks according to the *relative capacities* of the storage system's devices. Nevertheless, it is also possible to distribute data according to any other metrics devised by the system administrator. For instance, Random Slicing could be used to monitor I/O workload or power efficiency by simply defining an appropriate parameter to model the *relative I/O load* of each device or its *relative power consumption*.

By increasing or reducing these parameters, the administrator could control how many intervals are assigned to each storage device and thus redistribute the data load to other devices when I/O bottlenecks were detected or when a device needed to be spun down/turned off.

This kind of extensions are more dynamic in nature than a change in capacity and, as such, pose additional challenges like a more frequent update of the data structure in charge of partitions, or increased data migration. It remains to see, however, if Random Slicing's current techniques would be effective in such a dynamically changing environment but this falls beyond the scope of this thesis and is a subject of future research.

2.5 Methodology

Most previous evaluations of data distribution strategies are based on an analytical investigation of their properties. In contrast, we use a simulation environment to examine the real-world properties of the investigated protocols. The simulation environment has been developed through collaboration with researchers from the

University of Paderborn and the University of California Santa Cruz and is available online [98]. The software package also includes the parameter settings for the individual experiments described in this chapter.

First, we evaluate the scalability of Random Slicing with the different interval creation algorithms proposed in Section 2.4.240. This way, we determine how well the strategy scales and we select the best algorithm for the rest of the simulations.

Second, we run experiments for all distribution strategies described in Section 2.333 in an environment that scales from a few storage systems up to thousands of devices. We evaluate these strategies in terms of fairness, adaptivity, performance and memory consumption and compare the results with those of our proposal Random Slicing.

All experiments assume that each storage node (also called storage system in the following) consists of 16 hard disks (plus potential disks to add additional intra-shelf redundancy). Besides, most experiments distinguish between a homogeneous setting (all devices have the same capacity) and a heterogeneous settings (the capacity of devices varies).

In all experiments described, the implementations of Consistent Hashing, Share and Redundant Share use a custom implementation of the SHA1 pseudo-random number generator [41]. The RUSH* variants use the William-Hill generator, while the implementation of Random Slicing uses Thomas Wang's 64 bit mixing function [143].

2.6 Scalability of Random Slicing

As demonstrated by Theorem 2.4.138, Random Slicing's performance and memory consumption largely depends on how well the number of intervals scales when there are changes in the storage system. An excessive amount of new intervals can render the strategy useless due to memory or performance constraints. In this section we evaluate the different interval creation algorithms proposed in Section 2.4.240 and measure how well they scale in terms of performance, the number of new intervals created in each step, and the total number of intervals managed by the strategy.

2.6.1 Creation of New Intervals

Figure 2.3₄₆ plots the percentage of new intervals created when adding devices to the storage system. Each experiment begins with 50 devices and adds, in each step, as much new devices as needed to increase the overall capacity of the system by 10%, 50% or 80%, respectively. We evaluate a homogeneous setting where all the devices added have the same capacity, as well as a heterogeneous setting where the capacity of new devices increases by a factor of 1.5 in each step. In order to test long-term scalability, we continuously add devices until we surpass 25,000 devices.

Figures 2.3a, 2.3c and 2.3e46 depict the evaluation results for the homogeneous setting, while Figures 2.3b, 2.3d and 2.3f46 show the results for the heterogeneous setting. All four algorithms behave similarly in all experiments, which is to be expected as the number of intervals is not directly related to the capacity of the devices. All show an initial phase where adding devices leads to a high percentage of new intervals, with the Greedy algorithm even doubling the number of intervals in the first addition phase. This is not surprising since initially the number of intervals to work with is small, which limits the capability of the algorithms to create large gaps and increases the number of new intervals. This also explains why CutShift is more successful at reducing the number of new intervals than Greedy, since it is able to create larger gaps.

Assigning the largest partitions to the largest gaps also has some influence in reducing the number of new intervals, as the results for Greedy+Sorted and CutShift+Sorted show when comparing them against the non-sorted variants.

Note that once the algorithms have enough intervals to work with, they are significantly more effective: depending on the target increase in capacity, all strategies are able to reduce the number of new intervals to around 10%, 50% or 80% per step, respectively, and keep it steady even after more than 10 reorganizations. Interestingly enough, the number of new intervals created roughly corresponds to the increase in capacity, regardless of the number of devices added.

2.6.2 Absolute Intervals Created

Figure 2.447 shows the total number of intervals created by each algorithm. As expected, the CutShift algorithms produce a substantial reduction in the number of intervals when compared to the Greedy algorithms: by the end of each homogeneous experiment, CutShift algorithms produce ≈30,000, ≈12,000 and ≈9000 fewer intervals than Greedy algorithms (Figures 2.4a, 2.4c and 2.4e47, respectively). In

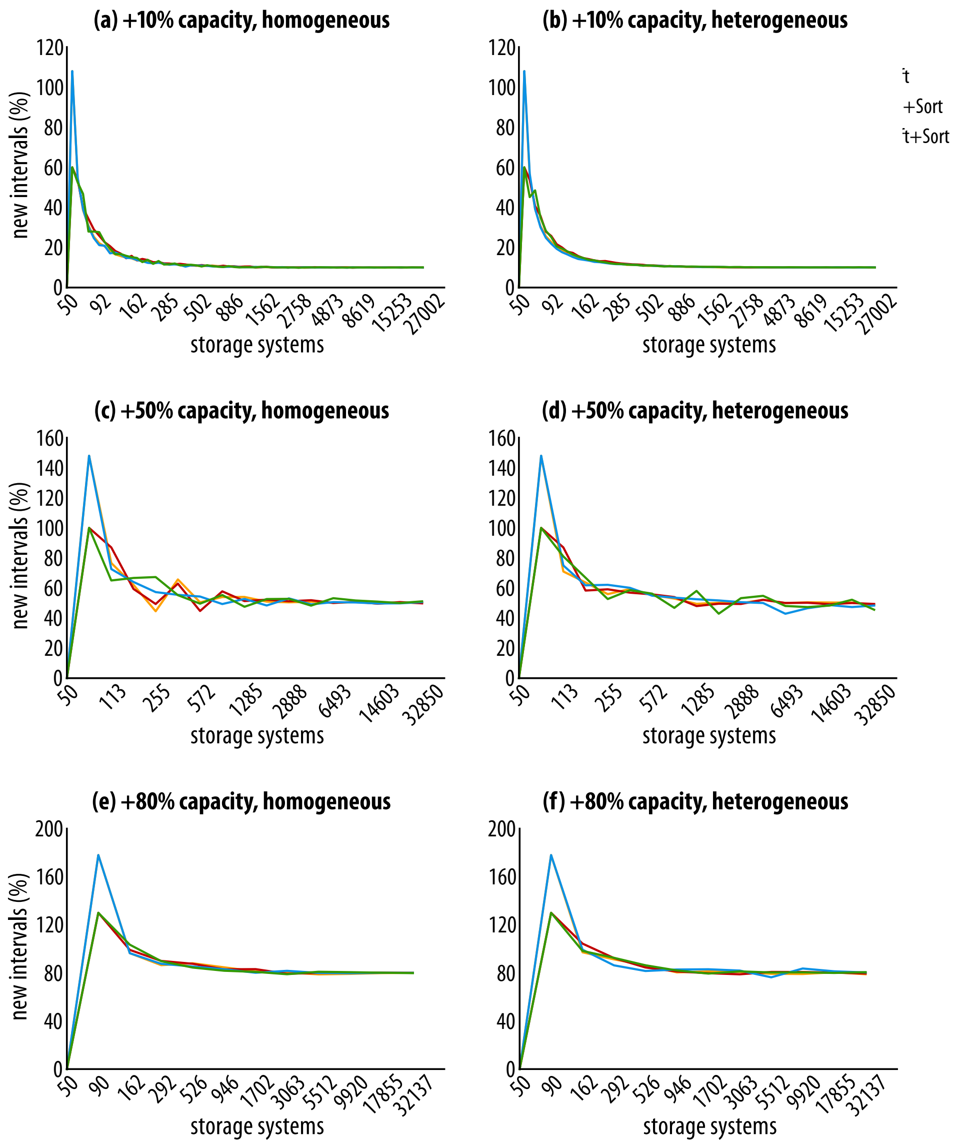


Figure 2.3: Percentage of new intervals created. Each step adds as many homogeneous or heterogeneous devices as necessary to reach the target capacity increase (i.e. +10%, +50%, and +80%). In the case of heterogeneous devices, the capacity of each device is increased by 1.5x.

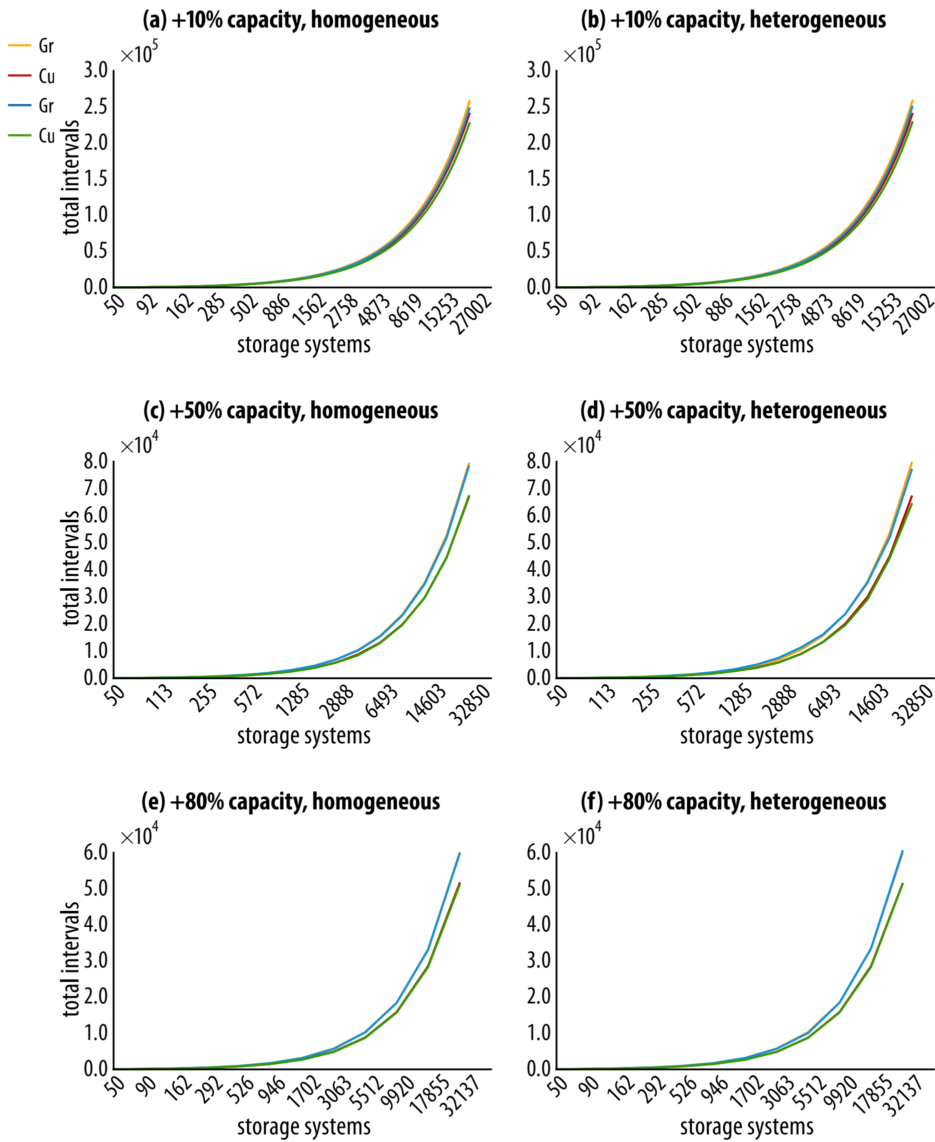


Figure 2.4: Total number of intervals created. Each step adds as many homogeneous or heterogeneous devices as necessary to reach the target capacity increase (i.e. +10%, +50%, and +80%). Again, for heterogeneous devices the capacity of each device is increased by 1.5x.

the heterogeneous experiments, this reduction amounts to ≈30,000, ≈15,000 and ≈8000 intervals (Figures 2.4b, 2.4d and 2.4f$_{47}$, respectively).

We also observe that, by the end of the experimental runs, the number of intervals grows considerably: for instance, the system shown in Figures 2.4a and 2.4b$_{47}$ needs 3×10^5 intervals after its 23rd reorganization. Such a number of intervals, though not insignificant, can be effectively managed using an appropriate *segment tree* [14] structure, which has a storage cost of $O(n \cdot log\ n)$ and a lookup cost of $O(log\ n)$ [37], where n is the number of intervals. Using this structure, the worst case memory requirement for 3×10^5 intervals is ≈167 MB,[2] which seems acceptable for over 25,000 devices and can be computed in a matter of minutes.

[2] *Assuming each node needs 32 bytes: 2 pointers for child trees and 2 integers for interval ranges.*

Based on the evaluation above, we select the CutShift+Sorted as our management algorithm, since it gives good results in the initial and the stable phase. From now on, in every experiment we discuss, Random Slicing will use this algorithm. For the sake of completion, Algorithm 2.1↷ shows the pseudo-code for this mechanism.

2.6.3 Performance Scalability

The graphs shown in Figure 2.5$_{50}$ demonstrate the evolution of Random Slicing's lookup time depending on the number of disks added to the system using the CutShift+Sorted assimilation algorithm. As in the previous experiments, we begin with 50 devices and add, in each step, as much devices as needed to increase the overall capacity of the system by 10%, 50% or 80%, respectively. For each configuration, we distribute 10^6 independent data blocks and measure the average time taken by Random Slicing in each operation. The measurements include 95% confidence intervals, and the homogeneous and heterogeneous environments use the same settings as in the previous experiments.

The results from these experiments show that the lookup time for a block is extremely fast, with a moderate growth trend that exhibits a lookup penalty of ≈1.5 µs in the latest configurations, which use 27,002, 49,272, and 57,845 devices, respectively (see Figures 2.5a to 2.5f$_{50}$). This moderate growth of the lookup time is due to two reasons: first, the tree structure used in our implementation is able to model the configuration of intervals using very few levels, which in practice turn the log I times to $O(1)$; second, the lookup time depends on the number of intervals used by the strategy rather than the number of devices. Thus, the better the assimilation algorithm can contain the growth of new intervals, the better the results produced by the lookup structure. All in all, these results show that Random Slicing is able to handle large numbers of devices with small performance and memory overhead.

Algorithm 2.1: CutShift Gap Collection Algorithm

Require: $Bins = \{b_0, \dots, b_{n-1}, b_n, \dots, b_{p-1}\}$ such that $b_0, \dots b_{n-1}$ are the capacities for old bins and $b_n \dots b_{p-1}$ are the capacities for new bins
Require: $Intervals = \{I_0, \dots, I_{q-1}\}$
Require: $(p > n) \wedge (q \geq n)$
Ensure: $Gaps = \{G_0, \dots, G_{m-1}\}$

```
 1: procedure CutShift(Bins, Intervals, Gaps)
 2:     ∀i ∈ {0, …, n − 1} : c_i ← b_i / Σ_{j=1}^{n−1} b_j
 3:     ∀i ∈ {0, …, p − 1} : č_i ← b_i / Σ_{j=1}^{p−1} b_j
 4:     ∀i ∈ {0, …, n − 1} : r_i ← BlockCount(c_i − č_i)
 5:     Gaps ← ∅
 6:     for all i ∈ Intervals do
 7:         b ← bin assigned to i
 8:         g ← last gap from Gaps
 9:         if r_b > 0 then
10:             if Length(i) < r_b then
11:                 if Adjacent(g, i) then
12:                     g.end ← g.end + Length(i)
13:                 else
14:                     Gaps ← Gaps + {i.start, i.end}
15:                 end if
16:                 r_b ← r_b − Length(i)
17:                 if last interval was assimilated completely then
18:                     CutIntervalEnd ← False
19:                 end if
20:             else
21:                 if Adjacent(g, i) then
22:                     g.end ← g.end + Length(i)
23:                 else
24:                     if CutIntervalEnd then
25:                         Gaps ← Gaps + {i.end − r_b, i.end}
26:                     else
27:                         Gaps ← Gaps + {i.start, i.start + r_b}
28:                     end if
29:                 end if
30:                 r_b ← 0
31:                 CutIntervalEnd ← ¬CutIntervalEnd
32:             end if
33:         end if
34:     end for
35: end procedure
```

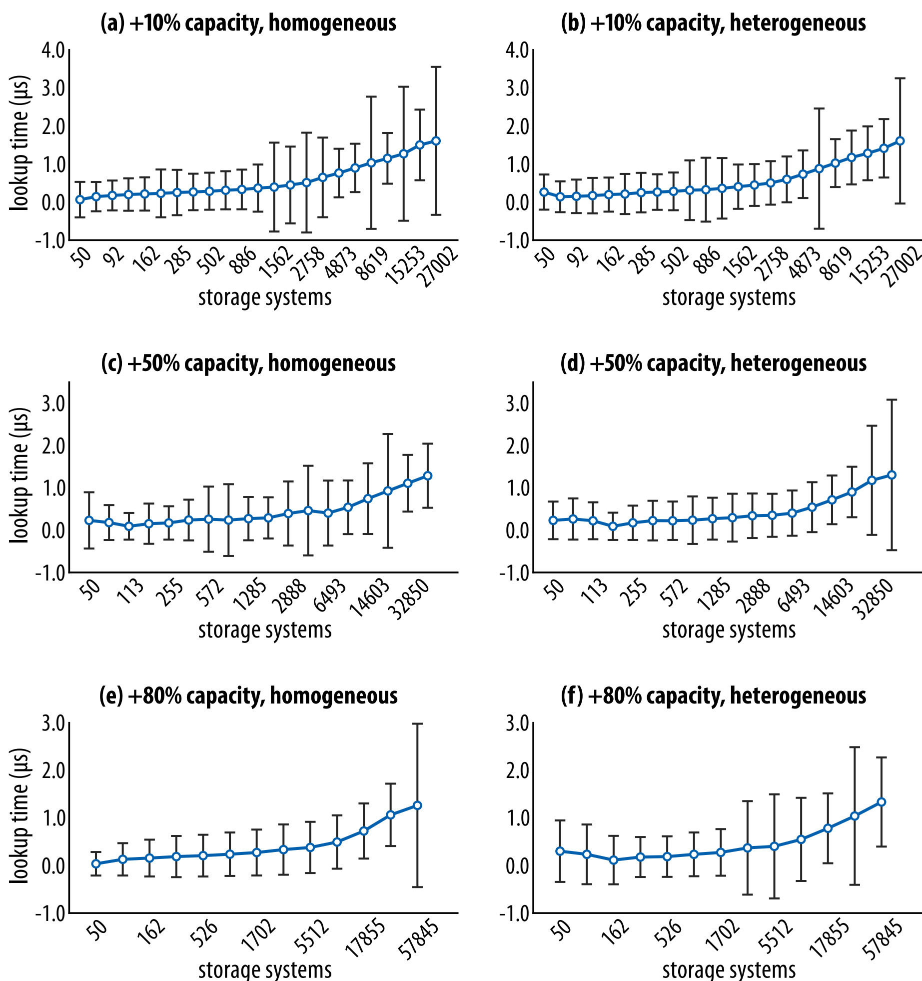


Figure 2.5: Lookup time after several reorganizations. Each step adds as many homogeneous or heterogeneous devices as necessary to reach the target capacity increase (i.e. 10%, 50%, and 80%). In the case of heterogeneous devices, the capacity of each device is increased by 1.5x. Each experiment distributes 10^6 data blocks and measures the time spent by Random Slicing for each individual lookup. Each measure includes 95% confidence intervals.

2.7 Comparative Evaluation

The following section evaluates the impact of the different distribution strategies on the data distribution quality, the memory consumption of the different strategies, their adaptivity and their performance.

2.7.1 Experimental Setup

We assume that each storage system in the homogeneous setting can hold up to $k \cdot 500{,}000$ data items, where k is the number of copies of each block. Assuming a hard disk capacity of 1 TB and putting 16 hard disks in each shelf means that each data item has a size of 2 MB. The number of placed data items is $k \cdot 250{,}000$-times the number of storage systems. In all cases, we compare the fairness, the memory consumption, as well as the performance of the different strategies for a different number of storage systems.

In the heterogeneous setting we begin with 128 storage systems and we add 128 new systems in each step, each with 1.5-times the size of the previously added system. We are placing again half the number of items, which saturates the capacity of all disks. For each of the homogeneous and heterogeneous tests, we also count the number of data items, which have to be moved in case we are adding disks, so that the data distribution delivers the correct location for a data item after the redistribution phase. The number of moved items has to be as small as possible to support dynamic environments, as the systems typically tend to a slower performance during the reconfiguration process.

We will show that the dynamic behavior can be different if the order of the k copies is important, e.g., in the case of RAID parities, Reed-Solomon, or EvenOdd erasure codes, or if this order can be neglected in the case of pure replication strategies [108, 17, 33].

All graphs presented in the section contain four bars for each number of storage systems, which represent the experimental results for one, two, four, and eight copies (please see Figure 2.652 for the color codes in the legend). The white boxes in each bar represent the range of results, i.e., between the minimum and the maximum usage. Also, in some cases the white boxes include errorbars to depict the standard deviation measured for the experiments. Small or non-existing white boxes indicate a very small deviation between the different experiments.

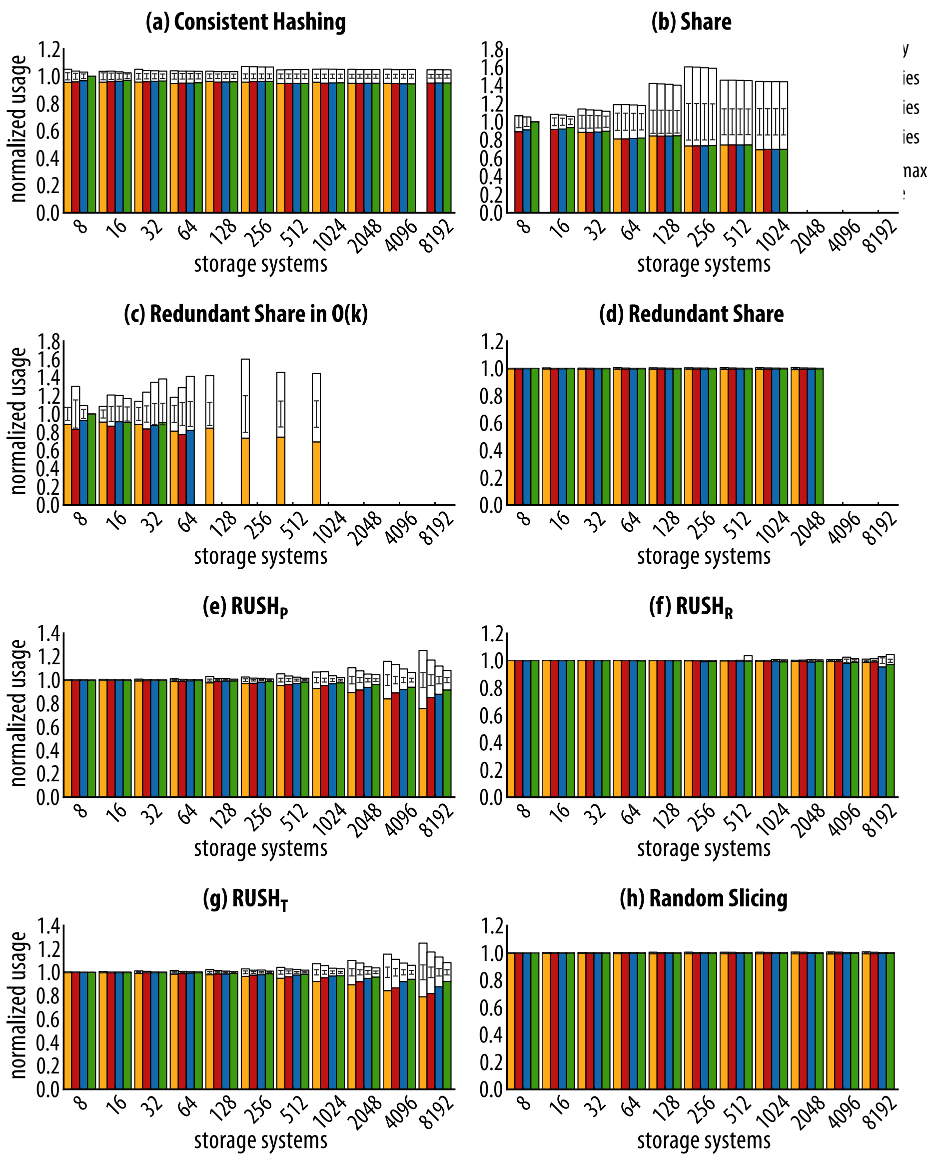


Figure 2.6: Scalability of the fairness achieved by each strategy in homogeneous settings. Each storage system has a capacity of $k \cdot 500{,}000$ data items and the experiment allocates $k \cdot n \cdot 250{,}000$ data items, where k is the number of copies used in the experiment and n the number of storage systems. A normalized usage of 1.0 represents an ideally balanced data distribution.

2.7.2 Fairness

The first simulations evaluate the fairness of the strategies for different sets of homogeneous disks, beginning at 8 storage systems and going up to 8192 storage systems (see Figure 2.6↶). Notice that there are some missing results for Consistent Hashing, Share and Redundant Share. These correspond to configurations that took too much time to evaluate or used more resources than available.

Consistent Hashing was developed to evenly distribute one copy over a set of homogeneous disks of the same size. Figure 2.6a↶ shows that the strategy is able to fulfill these demands for the test case, in which all disks have the same size. The difference between the maximum and the average usage is always below 7% and the difference between the minimum and average usage is always below 6%. The deviation is nearly independent from the number of copies as well as from the number of disks in the system, and therefore the strategy can be reasonably well applied. We have thrown $400 \cdot \log n$ points for each storage system (please see Section 2.3.1$_{34}$ for the meaning of points in Consistent Hashing).

The fairness of Consistent Hashing can be improved by throwing more points for each storage system (see Figure 2.7$_{54}$ for an evaluation with 64 storage systems). The evaluation shows that initial quality improvements can be achieved with very few additional points, while further small improvements require a high number of extra points per storage system. Note that, $400 \cdot \log n$ points represent 2400 points for $n = 64$ storage systems, which means that the strategy already needs a high number of points, and further quality improvements will become very costly.

Share was developed to overcome the drawbacks of Consistent Hashing for heterogeneous disks. Its main idea is to (randomly) partition the disks into intervals and assign a set of disks to each interval. Inside an interval, each disk is treated as homogeneous and strategies like Consistent Hashing can be applied to finally distribute the data items. This basic idea implies that Share has to compute and keep the data structures for each interval. Note that 1000 disks lead to a maximum of 2000 intervals, implying 2000-times the memory consumption of the applied uniform strategy. On the other hand, the number of disks inside each interval is smaller than n, which is the number of disks in the environment. The analysis of Share shows that on average $c \cdot \log n$ disks participate in each interval (see Section 2.3.2$_{34}$; without loss of generality we will neglect the additional $\frac{1}{\delta}$ to keep the argumentation simple). Applying Consistent Hashing as the homogeneous strategy thus leads to a memory consumption which is in $O(n \cdot \log^2 n \cdot \log^2(\log n))$, only factor of $\log^2(\log n)$ bigger than the memory usage of the original Consistent Hashing.

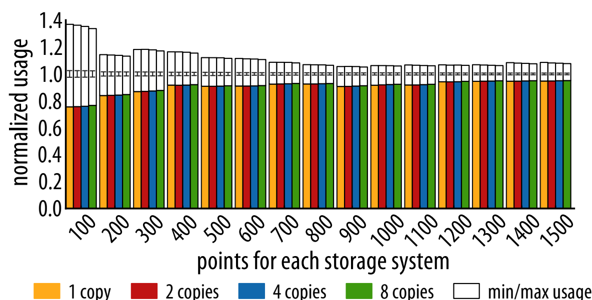


Figure 2.7: Evaluation of the impact of using different number of points in the fairness offered by the strategy Consistent Hashing, for 64 homogeneous storage systems. Notice that a normalized usage of 1.0 represents an ideally balanced data distribution.

Unfortunately, it is not possible to neglect the constants in a real implementation. Figure 2.6b52 shows the fairness of Share for a stretch factor of $3 \cdot \log n$, which shows huge deviations even for homogeneous disks. A deeper analysis of the Chernoff-bounds used by the authors [24], shows that it would have been necessary to use a stretch factor of 2146 to keep fairness in the same order as that achieved with Consistent Hashing, which is infeasible in scale-out environments.

Simulations including different stretch factors for 64 storage systems for Share are shown in Figure 2.8↷, where the x-axis depicts the stretch factor divided by $\ln n$. The fairness can be significantly improved by increasing the stretch factor. Unfortunately, a stretch factor of 32 already requires in our simulation environment more than 50 GB main memory for 64 storage systems, making Share impractical in bigger environments. For this reason, in the following we will skip this strategy in our evaluations.

On the other hand,Redundant Share uses precomputed intervals for each disk and therefore does not rely too much on randomization properties. The intervals exactly represent the share of each disk on the total disk capacity, leading to a very even distribution of the data items (see Figure 2.6d52). The drawback of this version of Redundant Share is that it has linear runtime, possibly leading to high delays in case of huge environments. Brinkmann et al. have presented enhancements, which enable Redundant Share to have a runtime in $O(k)$, where k is the number of copies [20]. Redundant Share in $O(k)$ requires a huge number of Share instances as sub-routines, making it impractical to support a huge number of disks and offer good fairness at the same time. Figure 2.6c52 shows that it is even difficult to support multiple copies for more than 64 disks, even if the required fairness is low, as 64 GB main memory have not been sufficient to calculate these distributions. Therefore, we will also neglect Redundant Share with runtime in $O(k)$ in the following measurements.

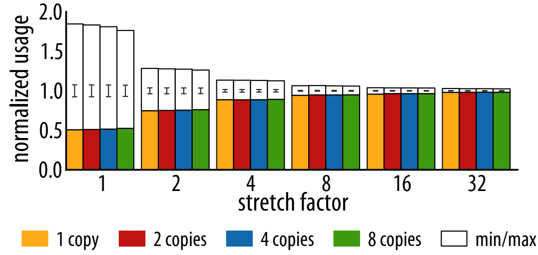


Figure 2.8: Evaluation of the fairness offered by the Share strategy using several stretch factors for 64 homogeneous storage systems. The x-axis depicts the stretch factor normalized by *ln*(*n*). Note that a normalized usage of 1.0 represents an ideally balanced data distribution.

RUSH_P, RUSH_T and RUSH_R distribute objects almost ideally according to the appropriate weights, though the distribution begins to degrade as the number of disks grows (see Figures 2.6e to 2.6g$_{52}$). Interestingly, however, this deviation from the ideal load decreases when the number of copies increases, which might imply that the hash function is not as uniform as expected and needs more samples to provide an even distribution.

In Random Slicing, precomputed partitions are used to represent a disk's share of the total system capacity, in a similar way to Redundant Share's use of intervals. This property, in addition to the hash function used, enforces an almost optimal distribution of the data items, as shown in Figure 2.6h$_{52}$.

The fairness of the different strategies for a set of heterogeneous storage systems is depicted in Figure 2.9$_{56}$. As described in Section 2.7.1$_{51}$, we start with 128 storage systems and add every time 128 additional systems having 1.5-times the capacity of the previously added. Once again, missing results in Redundant Share and RUSH* are due to configurations too expensive in terms of the computing power available.

The fairness of Consistent Hashing in its original version is obviously very poor (see Figure 2.9a$_{56}$). Assigning the same number of points in the [0, 1)-interval for each storage system, independent of its size, leads to huge variations. Simply adapting the number of points based on the capacities leads to much better deviations (see Figure 2.9b$_{56}$). The difference between the maximum, minimum and the average usage is around 10% and increases slightly with the number of copies. In the following, we will always use Consistent Hashing with an adaptive number of copies, depending on the capacities of the storage systems.

Both Redundant Share and Random Slicing show again a nearly perfect distribution of data items over the storage systems, due to their precise modeling of disk capacities and the uniformity of the distribution functions (see Figures 2.9c and 2.9g$_{56}$, respectively).

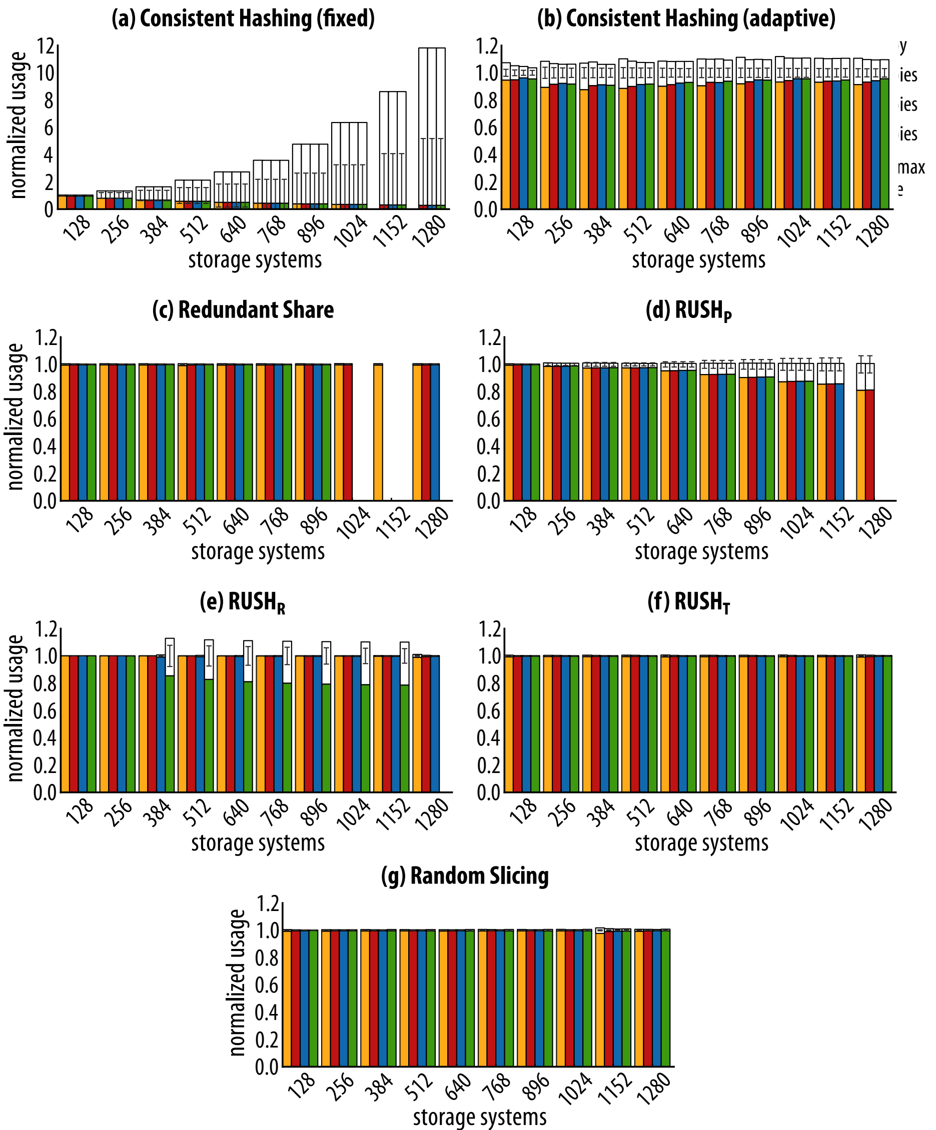


Figure 2.9: Scalability of the fairness achieved by each strategy in heterogeneous settings. Storage systems in the first configuration have a capacity of $k \cdot 500{,}000$ data items, where k is the number of copies used in the experiment. Subsequent configurations increase this capacity by 1.5x and each experiment allocates $1/2 \cdot$ TotalCapacity data items. A normalized usage of 1.0 represents an ideally balanced data distribution.

The fairness provided by $RUSH_P$ degrades steadily after the fourth reconfiguration. In contrast, $RUSH_T$ delivers a perfect data distribution, even when it was unable to do so in a homogeneous setting (see Figures 2.9d and 2.9f, respectively). Figure 2.9e, however, shows that $RUSH_R$ does a good distribution job for 1, 2, and 4 copies but degrades with 8 copies showing important deviations from the optimal distribution.

2.7.3 Memory Consumption and Compute Time

The memory consumption as well as the performance of the different data distribution strategies have a strong impact on the applicability of the different strategies. We assume that scale-out storage systems mostly occur in combination with huge cluster environments, where the different cores of a cluster node can share the necessary data structures for storage management. Assuming memory capacities of 192 GB per node in 2015 [6], we do not want to waste more than 10% (or approximately 20 GB) of this capacity for the metadata information of the underlying storage system. Furthermore, we assume access latencies of 5 ms for magnetic storage systems and access latencies of 50 µs for solid state disks. These access latencies set an upper limit on the time allowed for calculating the translation from a virtual address to a physical storage system.

As described in Section 2.7.151, all experiments begin with 128 storage systems and add 128 additional systems in every upgrade operation, each with 1.5-times the capacity of the previously added systems. During each test, we measure the memory used in each configuration as well as the performance of each request.

The bars in Figure 2.1058 represent the average allocated memory, and the white bars on top the peak consumption of virtual memory over the different tests. The points in that figure represent the average time required for a single request. These latencies include confidence intervals.

The memory consumption of Consistent Hashing only depends on the number and kind of disks in the system, while the number of copies k has no influence on it (see Figures 2.10a and 2.10b58). In Figure 2.10b58 we are throwing $400 \cdot \log n$ points for the smallest disk, and the number of points for larger disks grows proportionally to their capacity, which is necessary to keep fairness in heterogeneous environments. Using 1280 heterogeneous storage systems requires a memory capacity of nearly 9 GB, which is still below our limit of 20 GB.

The time to calculate the location of a data item only depends on the number of copies, as Consistent Hashing is implemented as a $O(1)$-strategy for a single copy.

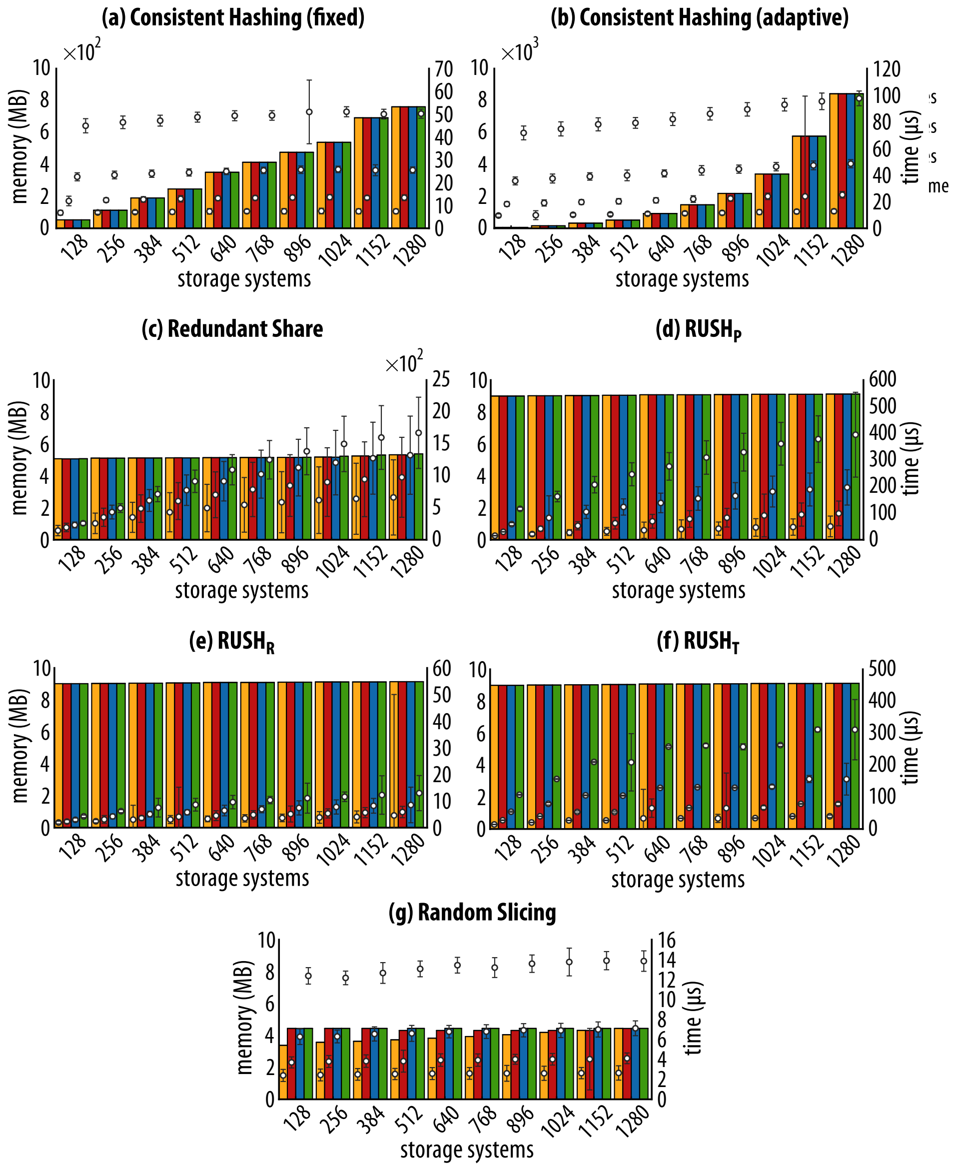


Figure 2.10: Scalability of the memory usage (left-hand y-axis) and the performance (right-hand y-axis) achieved by each strategy in a heterogeneous setting. The bars represent the average allocated memory for each configuration. The points represent the average response time for a single request (95% confidence interval).

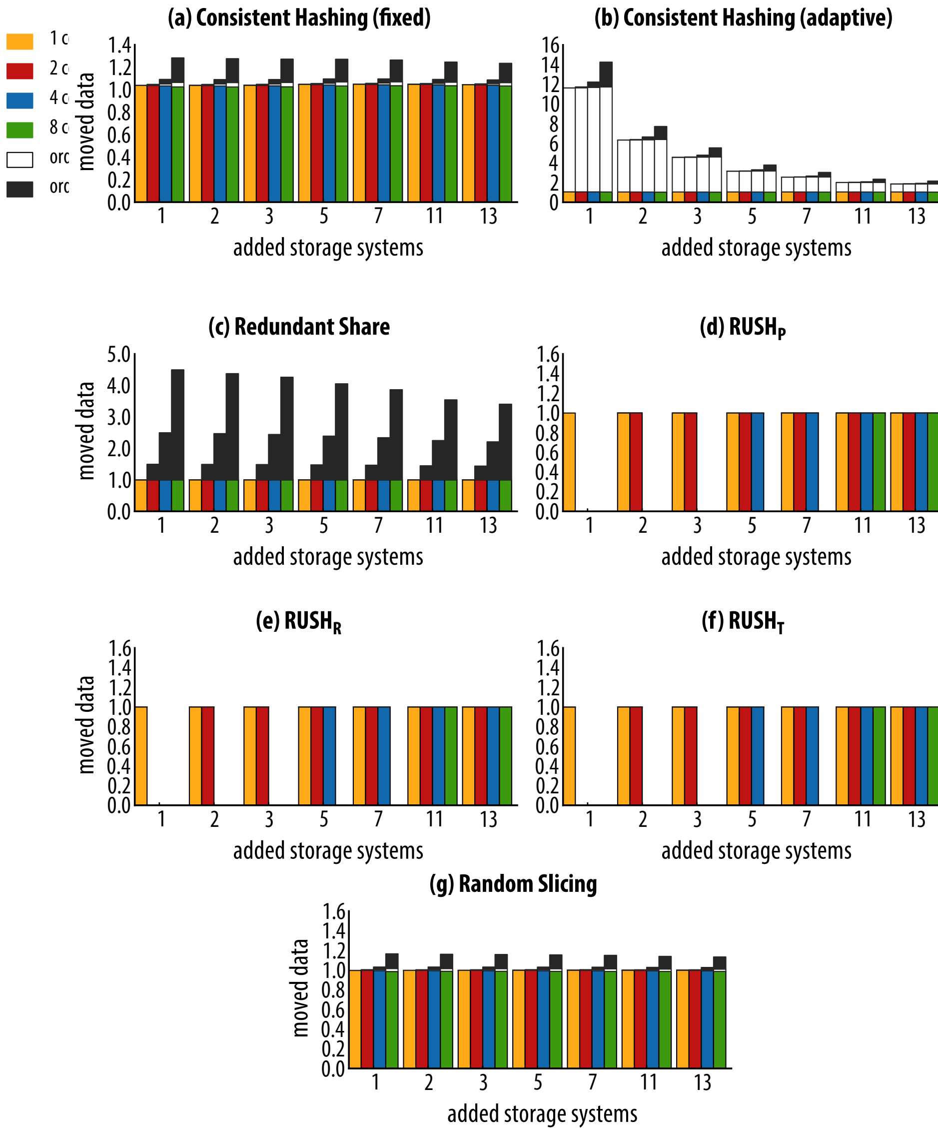


Figure 2.11: Scalability of the adaptivity achieved by each strategy in heterogeneous settings. All storage systems begin with $k \cdot 500{,}000$ data items and subsequent configurations increase this capacity by 1.5x, where k is the number of copies used in the experiment. A moved data value of 1.0 represents the minimum, ideal data movement.

The number of copies has only an influence for a small number of storage systems, i.e., it needs significantly more time to place 8 copies if it only uses 8 storage systems. The reason for this is how we chose to implement redundancy: we generate new random values into the $[0, 1)$ ring until we find k different disks, which can take quite long in case of hash collisions. Therefore, this proves that it is possible to use Consistent Hashing in scale-out environments based on solid state drives, as the average latency for the calculation of a single data item stays below 10 µs.

Redundant Share has very good properties concerning memory usage, but the computation time grows linearly with the number of storage systems. Even the calculation of a single item for 128 storage systems takes 145 µs. Using 8 copies increases the average access time for all copies to 258 µs, which is 50 µs for each copy, making it suitable for mid-sized environments that are based on SSDs. Increasing the environment to 1280 storage systems raises the calculation time almost linearly for a single copy to 669 µs, which is reasonable in magnetic disk based environments.

All RUSH variants show good results both in memory consumption and in computation time (see Figures 2.10d to 2.10f58), being RUSH_R the strategy with the lowest computation time. The reduced memory consumption is explained because the strategies do not need a great deal of in-memory structures in order to maintain the information about clusters and storage nodes. Lookup times depend only on the number of clusters in the system, which can be kept comparatively small for large systems.

Random Slicing shows very good behavior concerning memory consumption and computation time, as both only depend on the number of intervals I currently managed by the algorithm (see Figure 2.10g58). In order to compute the position of a data item x, the strategy only needs to locate the interval containing $f_B(x)$, which can be done in $O(\log I)$ using an appropriate tree structure. Furthermore, the algorithm strives to reduce the number of intervals created in each step in order to minimize memory consumption as much as possible. In practice, this yields an average access time of 5 µs for a single data item and 13 µs for 8 copies, while keeping a memory footprint similar to that of Redundant Share.

2.7.4 Adaptivity

Adaptivity to changing environments is an important requirement for data distribution strategies and one of the main drawbacks of standard RAID approaches. Adding a single disk to a RAID system typically either requires the replacement of

all data items in the system or splitting the RAID environment into multiple independent domains.

The theory behind randomized data distribution strategies claims that these strategies are able to compete with a best possible strategy in an adaptive setting. This means that the number of data movements to keep the properties of the strategy after a storage system has been inserted or deleted can be bounded against the best possible strategy. We assume in the following that a best possible algorithm just moves as much data from old disks to new disks (and respectively from removed disks to remaining disks), as necessary to have the same usage on all storage systems. All bars in Figure 2.11$_{59}$ have been normalized to this definition of an optimal algorithm.

Furthermore, we distinguish between placements where the ordering of the data items is relevant and where it is not. The first case occurs, e.g., for standard parity codes, where each data item has a different meaning (labeled "order kept" in Figure 2.11$_{59}$). If a client accesses the third block of a parity set, then it is necessary to receive exactly that block. In contrast, the second case occurs for RAID-1 sets, where each copy has the same content and receiving any of these blocks is sufficient (labeled "order changed"). We will see in the following that not having to keep the order strongly simplifies the rebalancing process.

We start our tests in all cases with 128 storage systems and increase the number of storage systems by 1, 2, 3, 5, 7, 11, or 13 storage systems. The new storage systems have 1.5-times the capacity of the devices in the previous configuration.

The original Consistent Hashing paper shows that the number of replacements is optimal for Consistent Hashing by showing that, in a homogeneous setting, data is only moved from old disks to new disks in case of the insertion of a storage system or from a removed disk to old disks in case of a deletion [70]. Figure 2.11b$_{59}$ shows a very different behavior, the number of data movements is sometimes more than 20-times higher than necessary. The reason is that we are placing $400 \cdot \lceil \log n \rceil$ points for each storage system and $\lceil \log n \rceil$ increases from 7 to 8 when adding storage system number 129. This leads to a large number of data movements between already existing storage systems. Furthermore, the competitiveness strongly depends on whether the ordering of the different copies has to be maintained or can be safely ignored.

Figure 2.11a$_{59}$ shows the adaptivity of Consistent Hashing in case that the number of points is fixed for each individual storage system and only depends on its own capacity. We use 2400 points for the smallest storage system and use a proportionally higher number of points for bigger storage systems. In this case the

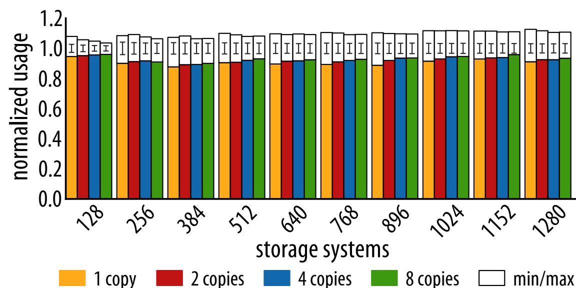


Figure 2.12: Evaluation of the fairness achieved by Consistent Hashing when using a fixed number of points for each storage system in a heterogeneous environment. The first 128 storage systems begin with 2400 points, which increase proportionally with the capacity of the newly added devices.

insertion of new storage systems only leads to data movements from old systems to new ones (and not between old ones) and therefore the adaptivity is very good in all cases. Figure 2.12↑ shows that the fairness in this case is still acceptable even in a heterogeneous setting.

The adaptivity of Redundant Share for adding new storage systems is nearly optimal, which is in line with the analysis previously presented [21]. Nevertheless, Redundant Share is only able to achieve an optimal competitiveness if a new storage system is inserted that is at least as big as the previous ones. Otherwise it can happen that Redundant Share is only log n-competitive (see Figure 2.11c59).

Figure 2.1159 shows that RUSH variants adapt nearly optimally when storage nodes are added. Note, however, that we did not evaluate the effect on replica ordering because the current implementations do not support replicas as distinct entities. Instead, RUSH variants distribute all replicas within one cluster. Note also that the missing columns in the results correspond to configurations not accepted in the current implementation.

Figure 2.11g59 shows that the adaptivity of Random Slicing is very good in all cases. This is explained because intervals for new storage systems are always created from fragments of old intervals, thus forcing data items to migrate only to new storage systems.

2.7.5 Summary

We conclude this section with a brief qualitative overview of the results collected. Table 2.2↷ summarizes our observations on the evaluated strategies, for all the properties examined. Where available, we provide either average or worst case values of every parameter examined, in order to offer a general view of each strategy's strong points and weaknesses.

Table 2.2: Properties of the examined strategies in heterogeneous architectures

Strategy	Fairness	Memory usage	Lookup time	Adaptivity
Consistent Hashing (fixed)	Poor ($\delta \uparrow$ with n)	High ($\mu \approx 800$ MB)	Moderate ($\tau \approx 50$ µs)	Good ($\alpha \approx 7\%$)
Consistent Hashing (adapt.)	Moderate ($\delta \approx 10\%$)	High ($\mu \approx 8$ GB)	High ($\tau \approx 98$ µs)	Poor ($\alpha \approx 1172\%$)
Redundant Share	Good ($\delta \approx 0.36\%$)	Low ($\mu \approx 5$ MB)	Very High ($\tau \approx 1800$ µs)	Good ($\alpha \approx 0.08\%$)
$RUSH_P$	Poor ($\delta \uparrow$ with n)	Low ($\mu \approx 9$ MB)	Very High ($\tau \approx 400$ µs)	Very Good[1] ($\alpha \approx 0.001\%$)
$RUSH_R$	Good, if k<8 ($\delta \approx 0.22\%$) Poor, if k=8 ($\delta \uparrow$ with n)	Low ($\mu \approx 9$ MB)	Low ($\tau \approx 14$ µs)	Very Good[1] ($\alpha \approx 0.001\%$)
$RUSH_T$	Good ($\delta \approx 0.36\%$)	Low ($\mu \approx 9$ MB)	Very High ($\tau \approx 300$ µs)	Very Good[1] ($\alpha \approx 0.05\%$)
Random Slicing	Good ($\delta \approx 0.4\%$)	Low ($\mu \approx 4.5$ MB)	Low ($\tau \approx 14$ µs)	Good ($\alpha \approx 1.63\%$)

Definitions used: n, number of devices; k, number of copies; δ, average deviation from ideal load; μ, worst case memory consumption; τ, worst case lookup time; α, worst case deviation from ideal number of movements.

[1] The implementation evaluated does not support replicas as distinct entities.

2.8 Conclusions

In this chapter of the thesis, we have shown that many randomized data distribution strategies are unable to scale to Exascale environments, since either their memory consumption, their load deviation, or their processing overhead is too high. Nevertheless, they are able to easily adapt to changing environments, a property which cannot be delivered by table or rule-based approaches.

The proposed Random Slicing strategy combines the advantages of all these approaches by keeping a small table and thereby reducing the amount of necessary random experiments. The presented evaluation and comparison with well-known strategies shows that Random Slicing is able to deliver the best fairness in all the cases studied and to scale up to Exascale data centers.

Chapter 3

PRNGs in Data Distribution

> *"Random numbers should not be generated with a method chosen at random."*
> — Donald E. Knuth

> *"Any one who considers arithmetical methods of producing random digits is, of course, in a state of sin."*
> — John von Neumann

3.1 Motivation

Chapter 229 shows that one of the most efficient and effective ways to achieve a balanced data load is using hashing techniques to distribute data. This relies on the intrinsic connection that exists between randomness and evenness of distribution: since a random number sequence is defined as a series of numbers that are drawn from a set of *equally-probable values*, and where each draw is *statistically independent* from the others, it can be safely considered that such a sequence follows a uniform distribution [11, 114, 56].

In practice, however, real randomness cannot be (easily) achieved using a computer, as they have the odd inclination of blindly following their programmed instructions, which makes them completely predictable. For this reason, two main approaches are used in order to generate random numbers for computer programs: Pseudo-Random Number Generators (PRNGs) and True Random Number Generators (TRNGs).

PRNGs are algorithms that use mathematical formulae or precomputed tables to produce a sequence of numbers that *appears* random. PRNGs typically are *efficient* (they can produce many numbers in a small amount of time) and *deterministic* (a sequence can be reproduced if the starting point is known) which makes them extremely useful for simulation and modeling applications. Usually, they are also *periodic*, i.e., the sequence eventually repeats itself. Although periodicity is undesirable in a truly random sequence, modern PRNGs have a really long period that can be ignored for most purposes.

TRNGs, on the other hand, normally extract randomness from physical phenomena like radioactive decay [142], atmospheric noise [56] or even snapshots from lava lamps [90], and feed it to a computer. Notably, this produces a true, non-periodic and non-deterministic, random sequence as long as the source is carefully selected. Compared to PRNGs, however, TRNGs generally need a considerably longer time to generate numbers, which makes them *inefficient* for applications with stringent performance constraints.

Therefore, even though using true randomness would be desirable in a large-scale storage system, the performance constraints associated to TRNGs force us to discard them and rely on PRNGs in order to construct the required hashing functions. For this reason, it is very important to choose an appropriate PRNG in order to design an effective hashing function. Specifically, the PRNG chosen to implement the hashing function h has two main requirements:

1. The computation time of $h(x)$ for each data value x should be as small as possible. Since the PRNG is an integral part of h its efficiency is of the utmost importance.

2. Successive calls to $h(x_i)$ should distribute input data as evenly as possible, for each data value x_i. This means that the sequence of numbers produced by the PRNG must be as uniformly distributed as possible.

The main contribution of this chapter is to evaluate how PRNGs impact data distribution in randomized strategies. Most specifically, we analyze the influence (or lack thereof) of eighteen different pseudo-random number generators in all the strategies studied in Chapter 229. We evaluate each PRNG individually using well-known randomness tests, and we also measure the quality of distribution and the performance it offers to each strategy. We believe that the results of this analysis may help other researchers choose an appropriate PRNG when designing and implementing new randomized data distribution strategies.

3.2 Introduction to (Pseudo-)Randomness

Formally, a PRNG is defined as a deterministic algorithm that generates a sequence of numbers $X_n = \{u_0, u_1, \ldots, u_n\}$ that approximates the properties of truly random numbers. According to NIST,[1] a random bit sequence can be interpreted *as the result of the flips of an unbiased "fair" coin with sides that are labeled "0" and "1", with each flip having a probability of exactly 1⁄2 of producing a "0" or "1". Furthermore, the flips are independent of each other: the result of any previous coin flip does not affect future coin flips* [122].

[1] *National Institute of Standards and Technology, U.S. Department of Commerce: federal agency that works with industry to develop and apply technology, measurements, and standards.*

Obviously, numbers generated deterministically cannot be truly random, but it is usually sufficient that finite segments of the sequence behave in a manner indistinguishable from an actual random sequence. Nevertheless, since X_n is usually constructed based on a finite state (or "seed") it will eventually be periodic, i.e., there exists a positive integer p such that $u_{n+p} = u_n$ for a sufficiently large n. That minimal p is called the generator's *period* and, as we mentioned, it is important that it is much larger than the number of random numbers that will ever be used, to avoid the presence of patterns in X_n.

Depending on the mathematical approach used to generate the X_n sequence, pseudo-random number generators can be roughly classified in one of the following generic classes:

- *Linear congruential generators (LCGs)* are one of the oldest and best known PRNG algorithms [77]. They are very popular because they tend to be very efficient and easy to implement, and are included in the runtime libraries of various compilers. In these generators, the sequence of pseudo-random integers X_n is defined by the following recurrence:

  $$X_{n+1} \equiv (a \cdot X_n + c) \mod m, \tag{3.1}$$

  where X_0 is the *seed* or start value, $m > 0$ is the *modulus*, a is the *multiplier* $(0 < a < m)$ and c is an additive constant or *increment* $(0 \leq c < m)$.

  The main advantages of LCGs are that they are usually very fast and do not require much data to be stored in memory, but they are *extremely* sensitive to the values chosen for c, m and a [19]. In particular, it has been shown that poor choices of m, a and c can lead to large correlations between the values generated [113] (see Figure 3.168).

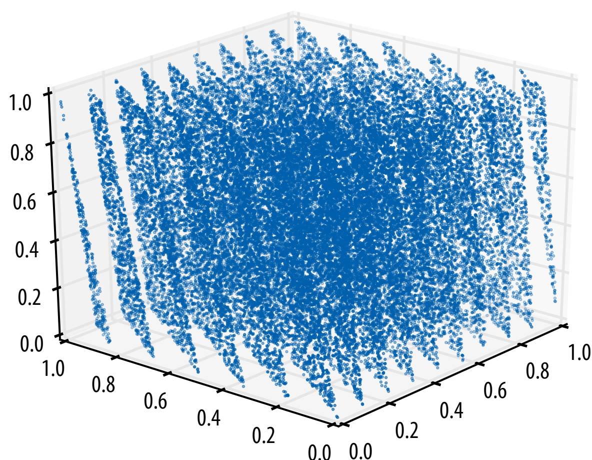


Figure 3.1: Spectral test for the RANDU generator: 3D scatter plot of the first 100,000 numbers generated by the linear congruential PRNG RANDU [66] ($m = 231$; $a = 65{,}539$; $c = 0$), normalized to the restricted interval [0.0, 1.0] and plotted as (x, y, z) successive triples. Notice the evident correlations between the numbers generated, which fall in 15 two-dimensional planes [55, 84].

Nevertheless, modern LCGs are capable of producing pseudo-random numbers of decent quality and, provided that c is nonzero, can have a maximal period if and only if: (1) c and m are relatively prime;[2] (2) $a - 1$ is divisible by all prime factors of m; and (3) $a - 1$ is a multiple of 4 [72, 129].

[2] $gcd(c, m) = 1$.

- *Multiplicative congruential generators (MCGs)* are a variant of LCGs that operate in a multiplicative group of integers modulo m. Hence, the sequence of pseudo-random integers X_n can be generally defined as:

$$X_{n+1} \equiv a \cdot X_n \mod m \tag{3.2}$$

where m is usually chosen to be either a prime number or a power of a prime number. The multiplier a is chosen to be an element of high multiplicative order[3] modulo m and the seed is relatively prime to m. They exhibit similar behavior and properties as standard LCGs.

[3] *The multiplicative order of a modulo m is the smallest positive integer k with* $a^k \equiv 1 \mod n$.

- *Inversive congruential generators (ICGs)* are a type of non-linear congruential PRNG algorithms that use the modular multiplicative inverse to generate

the sequence of values. The general formula for an ICG is the following:

$$\begin{cases} X_{n+1} \equiv (a \cdot X_n^{-1} + c) \mod m & \text{if } X_n \neq 0 \\ X_{n+1} = c & \text{if } X_n = 0 \end{cases} \tag{3.3}$$

Similarly to LCGs and MCGs, m is chosen to be a prime number for best results. Sequences of integers produced by ICGs have the advantage of being free of undesirable statistical deviations, and an algorithm is known [32] to maximize period length.

- *Lagged Fibonacci generators (LFGs)* are an improvement over standard LCGs that are based on the following generalization of the Fibonacci sequence:

$$X_{n+1} \equiv X_{n-r} \theta X_{n-s} \mod m, \tag{3.4}$$

for fixed "lags" r and s ($0 < r < s$) and where θ is some binary operator (addition, subtraction, multiplication or the bitwise arithmetic exclusive-or operator). For these generators, m is usually a power of 2. LFGs are very sensitive to initial conditions (e.g., it is required that at least one of the first s values chosen to initialize the generator be odd) and may show statistical defects in the output sequence if inappropriate parameters are used [155].

Note that when the recurrence used is of the form:

$$\begin{aligned} X_{n+1} &\equiv (X_{n-r} - X_{n-s} - c_{n-1}) \mod m, \\ c_n &= \begin{cases} 1 & \text{if } x_{n-s} - x_{n-r} - c_{i-q} < 0 \\ 0 & \text{if } x_{n-s} - x_{n-r} - c_{i-q} \geq 0 \end{cases} \end{aligned} \tag{3.5}$$

the LFGs are typically referred to as *Subtract with carry generators (SWCGs)* in literature [85].

- *Generalized feedback shift register (GFSR)* generators are defined by a map $f : F_q^d \rightarrow F_q^d$ of the form:

$$\begin{aligned} f(X_0, \ldots, X_{n-1}) &= (X_1, X_2, \ldots, X_n), \\ x_n &= C(X_0, \ldots, X_{n-1}) \end{aligned} \tag{3.6}$$

where $C : F_q^d \rightarrow F_q$ is a given linear function. The general idea is to use a *shift register*[4] to generate X_n from two or more previous numbers (called

[4] *An electronic device that shifts bits though a one-dimensional array.*

taps). Since the operation on previous values is deterministic, the next value of the sequence is completely determined by its current (or previous) state. Likewise, because the register has a finite number of possible states, it must eventually enter a repeating cycle. Depending on the intended application however, a well-chosen feedback function can produce a sequence of bits which appears random and which has a very long cycle.

Note that when C is of the form:

$$C(X_0, \dots, X_{n-1}) = a_0 \cdot X_0 + \dots + a_{n-1} \cdot X_{n-1} \tag{3.7}$$

for some given constants $a_i \in F_q$, the map is called a *linear feedback shift register (LFSR)*. The most commonly used linear function of single bits is the binary exclusive-or operation. The well-known Mersenne twister generator [86] is an example of a twisted GFSR.

- *Cryptographic hash functions.* A cryptographic hash function H is an algorithm that, given an arbitrary block of data, returns a cryptographic hash value h, such that any change to the data modifies h. That is, for any two distinct inputs A and B, $H(A)$ and $H(B)$ should be uncorrelated.

  Though not a PRNG *per se*, the output of a cryptographic function usually follows (by design) a uniform distribution over its period, which makes it theoretically suitable to generate sequence of pseudo-random numbers, albeit with more computation than classical PRNGs [82, 139].

In the following sections, we assess several PRNGs from each of these types in an attempt to find which are more likely to produce good results (both quality and performance-wise) when used for randomized data distribution.

3.3 Methodology

In order to evaluate the influence of PRNGs in randomized data distribution, we use the same simulation environment described in Section 2.543 in the previous chapter. This time however, we reevaluate each data distribution strategy, discarding its original PRNG implementation and substituting it with several PRNGs. By measuring the distribution quality and the performance offered by each strategy

Table 3.1: List of evaluated PRNGs

PRNG	Subclass
minstd_rand	LCG
minstd_rand0	LCG
ecuyer1988	LCG + LCG (additive combine)
kreutzer1986	LCG improved (shuffle)
hellekalek1995	ICG
rand48	MCG
Unix_rand	MCG
lagged_fibonacci607	LFG
ranlux3	LFG SWC (+ discard block)
ranlux64_3	LFG SWC (+ discard block)
ranlux3_01	LFG SWC (+ discard block)
ranlux64_3_01	LFG SWC (+ discard block)
mt11213b	GFSR (Mersenne twister)
mt19937	GFSR (Mersenne twister)
taus88	LFSR (XOR)
SHA1	Cryptographic hash
MD5	Cryptographic hash
Tiger-192	Cryptographic hash

Generator subclass: Linear congruential (LCG), inversive congruential (ICG) multiplicative congruential (MCG), lagged Fibonacci (LFG), linear feedback shift (LFSG), subtract with carry (SWC), generalized feedback shift register (GFSR), linear feedback shift register (LFSR).

during the simulations, we can assess which PRNGs influence data distribution significantly and whether any particular PRNG subclass is better (or worse) suited for data distribution.

The strategies we simulate are the same we discuss in Chapter 2_{29}, Consistent Hashing, Redundant Share, the three variants of RUSH and Random Slicing. Additionally, Table $3.1_{\uparrow}$ lists the set of pseudo-random number generators that we evaluate; we use implementations from Boost [36] and Libgcrypt [73], both well-known C/C++ libraries, since they are thoroughly and extensively tested. Besides, we try to cover a diverse set of PRNGs and, when possible, include more than one example of each particular subclass.

We simulate each strategy with all the aforementioned PRNGs and we measure performance and fairness in the same way as the experiments presented in Section 2.7_{51}. In these experiments, however, we distribute $m = 1000 \cdot n$ data blocks, with n being the number of configured storage systems. In order to avoid measurement artifacts as much as possible, all experiments have been run for 10, 100 and

1000 *homogeneous* storage systems, since, in the previous chapter, all the studied strategies showed almost ideal distributions in these environments.

Whenever we need to evaluate the quality of the numbers produced by each PRNG, we will resort to statistical tests. As we have seen, randomness is a rather abstract concept which is impossible to demonstrate, since, by definition, it cannot be expressed deterministically. Nevertheless, randomness can indeed be characterized and described in terms of probability, and it is for this reason that the quality of a pseudo-random sequence can be evaluated with statistical tests that compare it against a (statistically) true random sequence. Most of these tests initially assume that the sequence tested is random and try to assess the presence (or absence) of *patterns* that indicate if the sequence is random. In this chapter we will use two well-known batteries of statistical tests, the NIST [122] and the ENT [141] test suites.

3.4 Influence on Fairness

Figure 3.2 shows the results obtained when evaluating the fairness provided by each combination of PRNG and strategy, grouped by PRNG subclass. The white boxes in each bar represent the range of results between the minimum and maximum usage per disk, thus small or non-existing white boxes indicate a very small deviation from an ideal data distribution. Note that for some strategies we needed to split the y-axis to show all the relevant results, due to the huge variability of the measurements obtained with every algorithm.

Interestingly, the results are very different from those we saw in the previous chapter (see Section 2.7.2$_{53}$). Consistent Hashing, for instance, which showed a distribution that was close to ideal in the previous experiments, now exhibits a clearly unbalanced data distribution with some storage systems even receiving more than 250 times the expected amount of data blocks in some cases (see Figure 3.2a).

Other strategies seem to be less affected by the change of PRNG, though in general the evaluations show worse results than those obtained with the original hash functions, even for those strategies that previously showed a quasi-ideal distribution like Redundant Share, RUSH_R and Random Slicing (Figures 3.2b, 3.2d and 3.2f, respectively).

There does not seem to be much difference between the results for different subclasses as long as the number of disks is low. If we consider the generators individually, however, the results for all the PRNGS evaluated except for the crypto-

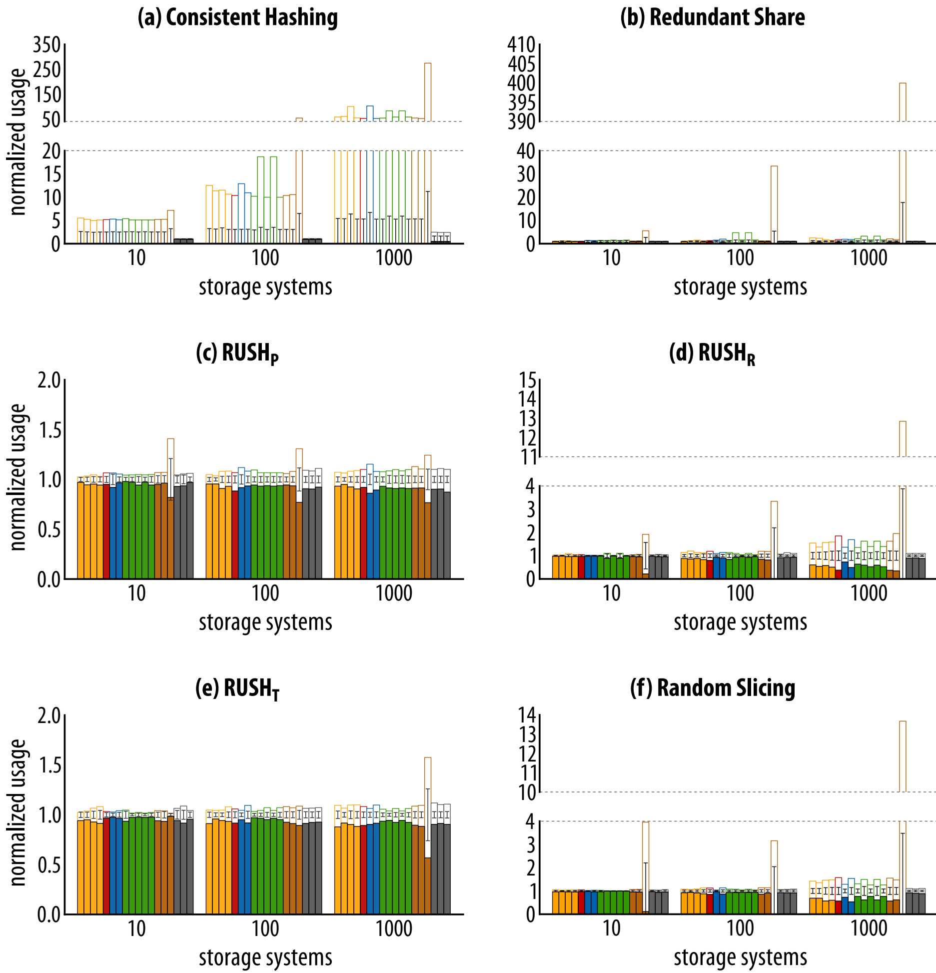


Figure 3.2: Influence of PRNG subclasses on fairness. Generators from left to right: LCGs: *minstd_rand, minstd_rand0, ecuyer1988, kreutzer1986*; ICGs: *hellekalek1995*; MCGs: *rand48, Unix_rand*; LFGs: *lagged_fibonacci607, ranlux3, ranlux64_3, ranlux3_01, ranlux64_3_01*; GFSR: *mt11213b, mt19937, taus88*; Cryptographic: *SHA1, MD5, Tiger-192*.

graphic functions (SHA1, MD5 and Tiger-192) degrade significantly when increasing the number of storage systems from 100 to 1000 devices. This behavior is less noticeable in Redundant Share (Figure 3.2b$_{73}$), because it uses $O(n)$ PRNG calls for each data block, which results in a better random distribution.

Most interestingly, the taus88 PRNG shows really bad results in all experiments, significantly degrading the fairness of all strategies when compared against the other PRNGs. In order to understand the reasons for this behavior, we must analyze the quality of the number distribution produced by each PRNG. Figure 3.3↷ shows a visualization of the first 2^{20} numbers produced by each PRNG in the previous experiment. Each number of the sequence is assigned an (x, y) coordinate in a 1024×1024 grid based on its order in the sequence, and a grayscale color based on its value. While this approach is not a formal or exhaustive one, it is a fast way to get a general impression of a generator's quality.

Surprisingly, most of the generators show obvious patterns in the distributions produced, even when considering only a few thousand generations. In particular, LF-based generators lagged_fibonacci607, ranlux3, ranlux64_3 ranlux3_01 and ranlux64_3_01 display highly predictable patterns that should not appear in a truly random sequence (Figures 3.3h to 3.3l↷, respectively). Also, notice how generator taus88 (Figure 3.3o↷) shows a clearly discernible pattern that explains the poor distribution quality seen in the previous experiment.

The reason for this poor distribution quality lies in how the examined strategies use PRNGs: in order to generate an appropriate hash value, strategies need to use the data block identifier as the seed of the generator, thus producing an independent X_n sequence for each block. Most PRNGs guarantee a certain level of randomness *within a sequence* but say nothing of the correlation between parallel sequences of different seeds, which can lead to more predictable distributions and poor fairness.

By contrast, generators kreutzer1986, hellekalek1995 and Unix_rand, as well as the Mersenne twister generators (mt11213b, mt19937) and the cryptographic hash functions (SHA1, MD5, Tiger-192) produce uniformly distributed values with no clear patterns, making them more suitable for data distribution (see Figures 3.3d, 3.3e, 3.3g, 3.3m, 3.3n and 3.3p to 3.3r↷, respectively). Note however that it has been shown [144, 133, 134] that both MD5 and SHA1 are not resistant to collision attacks.

For a more formal approach, we evaluate the same sequences of 2^{20} numbers with the ENT and NIST statistical batteries. For the ENT battery we show the results of each test compared against a reference value for a totally random sequence. For the NIST battery, however, we divide the sequence in 128 subsequences which

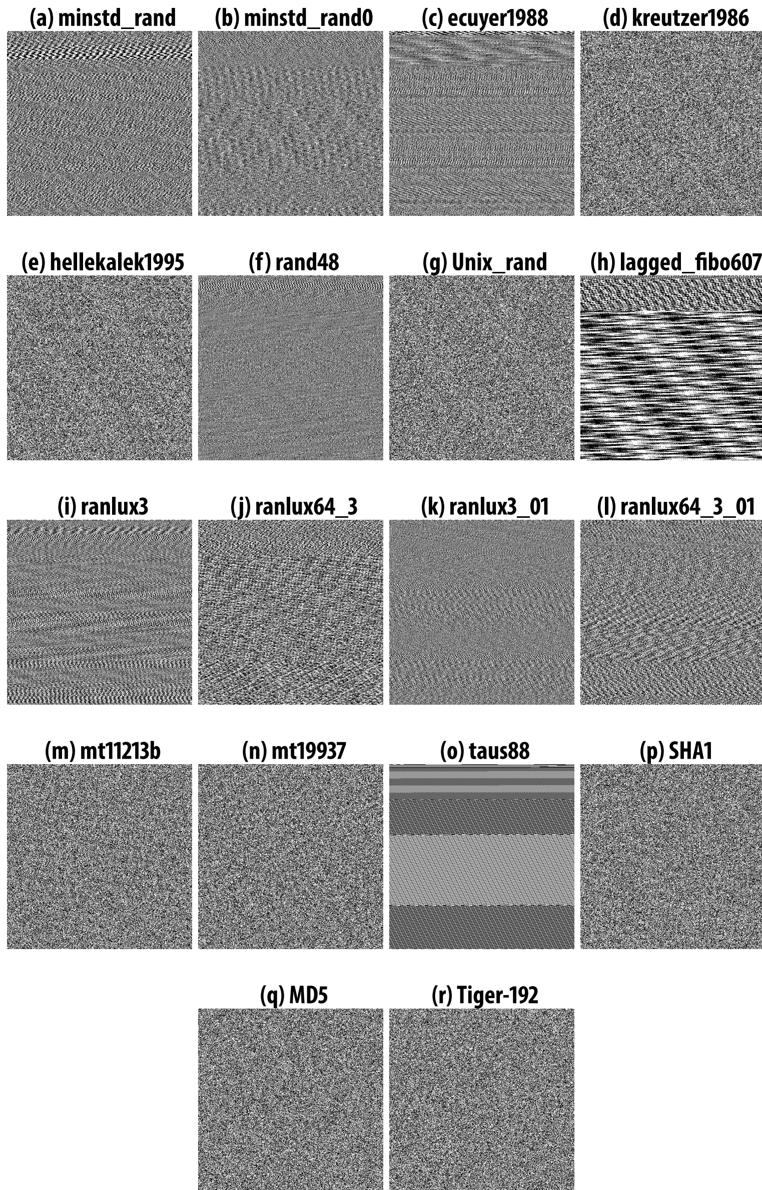


Figure 3.3: Test for patterns in the first 2^{20} numbers generated by several PRNGs. The experiment assigns an (x, y) coordinate to each number in the sequence and a grayscale color based on the value of the number produced. Note how some generators exhibit evident patterns that should not appear in a truly random sequence.

Table 3.2: Results obtained with the ENT suite

PRNG	Entropy (bits/byte)	χ^2 Statistic Distribution (p)	Mean deviation	Monte Carlo π estimation error	Serial correlation coefficient
minstd_rand	7.999	0.01%	0.049	0.131%	9.94E-4
minstd_rand0	7.999	0.01%	0.018	0.037%	2.83E-3
ecuyer1988	7.999	0.01%	0.055	0.080%	5.05E-4
kreutzer1986	7.999	0.01%	1.262	0.104%	3.94E-3
hellekalek1995	7.999	0.01%	0.146	0.050%	-7.42E-4
rand48	7.885	0.01%	0.031	0.669%	2.12E-4
Unix_rand	7.498	0.01%	7.034	7.090%	-4.86E-2
lagged_fibonacci607	7.996	0.01%	0.037	1.331%	1.43E-3
ranlux3	7.999	0.01%	0.078	0.091%	1.94E-4
ranlux64_3	7.996	0.01%	0.037	1.647%	1.43E-3
ranlux3_01	7.999	0.01%	0.078	0.091%	1.94E-4
ranlux64_3_01	7.996	0.01%	0.081	1.647%	-1.17E-3
mt11213b	7.999	0.01%	0.127	0.154%	5.87E-4
mt19937	7.999	0.01%	0.029	0.036%	-8.87E-4
taus88	7.865	0.01%	15.547	6.453%	-3.28E-2
SHA1	7.999	3.18%	0.029	0.036%	4.42E-4
MD5	7.999	71.33%	0.008	0.001%	-2.97E-4
Tiger-192	7.999	93.68%	0.032	0.014%	8.70E-5
Totally random	8	$5\% < p < 95\%$	0.0	0% (π)	0

are all subjected to the different tests. We show the proportion of successful subsequences and also list the specific tests where a certain subsequence failed. For the interested reader, we include a detailed discussion of each test and its interpretation in Appendix A_{141} (ENT battery) and Appendix B_{145} (NIST battery).

Table $3.2_{\uparrow}$ shows the results for the ENT evaluation. Most interestingly the χ^2 test classifies all distributions (except the cryptographic ones) as *almost certainly not random*, which confirms the visualizations shown in Figure 3.3_{75}. With respect to the other estimators, there are not clear evidences of which PRNGs behave better and which worse, and the results for generators of the same type seem unrelated. Cryptographic hashes, once again, display the best results for most tests, while taus88 generator does the contrary.

The results obtained with the NIST test battery (see Table $3.3_{\curvearrowright}$) show that only generators kreutzer1986, hellekalek1995, mt11213b, mt19937 and the cryptographic functions successfully pass all tests. Linear congruential generators minstd_rand, minstd_rand0 and ecuyer1988 show good results for over 73% of the subsequences

Table 3.3: Results obtained with the NIST suite

PRNG	Passed tests (proportion)	Failed tests
minstd_rand	0.867	FFT, Ent
minstd_rand0	0.933	FFT
ecuyer1988	0.733	Freq, CumSum, FFT, Serial
kreutzer1986	1.000	
hellekalek1995	1.000	
rand48	0.200	Freq, BFreq, CumSum, Runs, LRuns, Rank, FFT, NOT, OT, Uni, Ent, Serial
Unix_rand	0.267	Freq, BFreq, CumSum, Runs, LRuns, FFT, NOT, OT, Uni, Ent, Serial
lagged_fibonacci607	0.667	BFreq, Runs, LRuns, FFT, Ent
ranlux3	0.733	BFreq, FFT, Ent, Serial
ranlux64_3	0.533	BFreq, Runs, LRuns, FFT, OT, Uni
ranlux3_01	0.733	BFreq, FFT, Ent, Serial
ranlux64_3_01	0.533	BFreq, Runs, LRuns, FFT, OT, Uni
mt11213b	1.000	
mt19937	1.000	
taus88	0.133	Freq, BFreq, CumSum, Runs, LRuns, Rank, FFT, NOT, OT, Uni, Ent, Serial, LC
SHA1	1.000	
MD5	1.000	
Tiger-192	1.000	

FFT: Discrete Fourier Transform; Ent: Approximate Entropy; Freq: Frequency; CumSum: Cumulative Sums; BFreq: Block Frequency; LRuns: Longest Runs; Rank: Binary Matrix Rank; NOT: Non-Overlapping Template; OT: Overlapping Template; Uni: Maurer's Universal Statistical; LC: Linear Complexity

tested. On the contrary, however, multiplicative congruential generators rand48 and Unix_rand demonstrate poor results, with only ≈20% subsequences passing successfully. The taus88 generator shows the worst results once again, with only ≈10% good sequences.

These evaluations demonstrate that choosing a wrong PRNG can have disastrous balancing results for randomized data distribution, as it can invalidate all the benefits provided by a nonetheless good strategy.

3.5 Influence on Performance

Figure 3.478 shows the results obtained in the performance experiments for each combination of PRNG and strategy. Each bar shows the average time required for a single request. These times include upper 95% confidence intervals, which we plot as a white segment stacked atop the average bar.

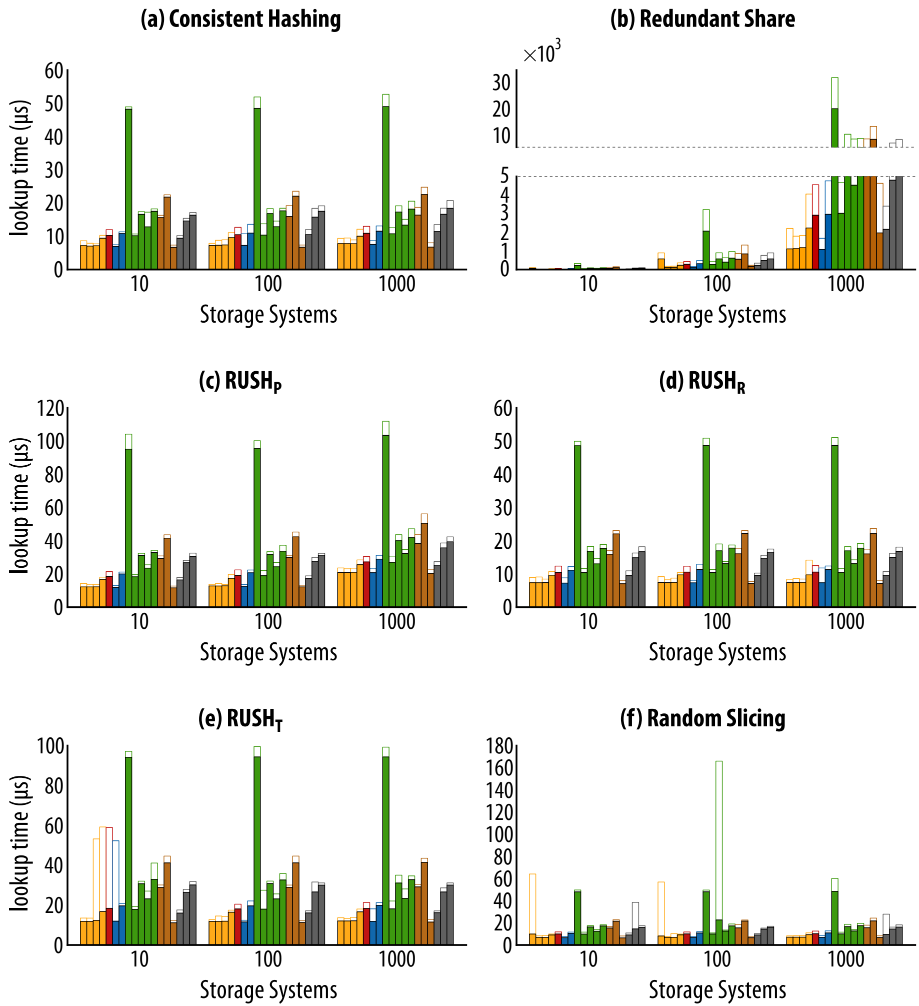


Figure 3.4: Influence of PRNG subclasses on performance. Generators from left to right: LCGs: *minstd_rand, minstd_rand0, ecuyer1988, kreutzer1986*; ICGs: *hellekalek1995*; MCGs: *rand48, Unix_rand*; LFGs: *lagged_fibonacci607, ranlux3, ranlux64_3, ranlux3_01, ranlux64_3_01*; GFSR: *mt11213b, mt19937, taus88*; Cryptographic: *SHA1, MD5, Tiger-192*.

Table 3.4: Final PRNG ranking

	Load distribution		Performance	
Position	**Generator**	**Subclass**	**Generator**	**Subclass**
1st	Tiger-192	Cryptographic	rand48	MCG
2nd	SHA1	Cryptographic	minstd_rand0	LCG
3rd	MD5	Cryptographic	ecuyer1988	LCG
4th	mt19937	GFSR (Mersenne)	minstd_rand	LCG
5th	hellekalek1995	ICG	taus88	LFSR
6th	mt11213b	GFSR (Mersenne)	SHA1	Cryptographic
7th	Unix_rand	MCG	kreutzer1986	LCG
8th	kreutzer1986	LCG	hellekalek1995	ICG
9th	lagged_fibonacci607	LFG	ranlux3	LFG SWC
10th	minstd_rand	LCG	Unix_rand	MCG
11th	minstd_rand0	LCG	ranlux3_01	LFG SWC
12th	ranlux64_3	LFG SWC	MD5	Cryptographic
13th	ranlux64_3_01	LFG SWC	ranlux64_3	LFG SWC
14th	ranlux3_01	LFG SWC	Tiger-192	Cryptographic
15th	ranlux3	LFG SWC	mt11213b	GFSR (Mersenne)
16th	ecuyer1988	LCG	ranlux64_3_01	LFG SWC
17th	rand48	MCG	mt19937	GFSR (Mersenne)
18th	taus88	LFSR	lagged_fibonacci607	LFG

Similarly to the previous experiments, there is a lot of variability in the results obtained. This time, however, the taus88 generator provides the best performance (on average) for all strategies except Consistent Hashing, where rand48 is slightly faster (see Figure 3.4a↶). Notice that in these experiments the lagged_fibonacci607 generator consistently shows the worst performance results, even though it was able to provide an acceptable fairness in the previous section.

Notice that all the PRNGs examined tend to behave similarly with 10, 100, or 1000 storage systems, which is to be expected since the generators should not be influenced by the number of devices in the system. There is one significant exception, however, with the Redundant Share strategy (Figure 3.4b↶), which shows increased response times in each successive setting. Once again, this can be explained because Redundant Share performs $O(k)$ calls to the selected generator for each data block, which can significantly increase the computation time if k is large.

Most interestingly, notice that PRNGs that provided similar levels of fairness in the previous experiment (e.g., SHA1, MD5, or Tiger), differ significantly when considering average response times. In addition, cryptographic hashes usually per-

form worse than simpler PRNGs that delivered similar fairness in the previous experiments, which can be a concern in time-critical conditions. This means that, when designing a randomized distribution strategy, a careful consideration of the selected PRNG is necessary in order to avoid potential performance pitfalls.

3.6 Evaluation Summary

To conclude the evaluation, and as a final contribution, we summarize the results of this chapter by deriving two rankings of the studied PRNGs (see Table 3.479). The first ranking orders PRNGs based on their effectiveness in distributing data load and the number of successful ENT/NIST tests passed, while the second one orders PRNGs based on the average performance they offer per request.

Based on this ranking, we conclude that cryptographic hash functions and Mersenne twister generators provide the best results concerning load balance, though they are not particularly fast when computing pseudo-random values. Nevertheless, the SHA1 generator shows good balancing results at acceptable performance costs, which makes it a good candidate for data distribution.

On the other hand, if distribution performance is an issue and some amount of unbalance is tolerable, linear congruential generators are among the fastest PRNGs in our evaluation with acceptable results for load balance.

3.7 Conclusions

In this chapter we have shown that it is extremely important to choose an appropriate PRNG when designing a randomized data distribution strategy. Such decision should be taken after careful evaluation of the PRNGs considered because, as we have seen, a badly chosen generator can degrade the quality of the strategy, effectively negating the benefits intended by the designer.

We have also demonstrated standard methods for evaluating PRNGs, and described some of the parameters that are most important for the quality of a pseudo-random sequence of numbers. Finally, we have ranked all the studied PRNGs based on their load distribution and performance results, which may serve future researchers to quickly assess the effectiveness of each of the studied generators.

Chapter 4

Long-term Locality in Mass Storage

"There are two kinds of truths: those of reasoning and those of fact. The truths of reasoning are necessary and their opposite is impossible; the truths of fact are contingent and their opposites are possible."
— Gottfried Wilhelm von Leibniz

4.1 Motivation

As we discuss in Chapter 2_{29}, pseudo-randomized distribution is an effective way to store data efficiently that reorganizes only the *minimum amount of data* necessary to maintain a balanced load. Nevertheless, even this minimum reorganization of data can be prohibitively expensive in Petascale and Exascale storage systems.

Figure 4.1_{84} shows an optimistic estimation of the time required to reorganize existing data with current technology[1] due to infrastructure upgrades. It clearly shows that, when stored data grows to 1 Petabyte or more, the associated reorganization time spirals upwards, rapidly making it cost-prohibitive. And that is assuming an scenario where we can shutdown the storage infrastructure at will while disregarding client applications, which is completely unacceptable in 24/7 environments!

[1] *As of July 2013.*

In this situation, it is obvious that some measures have to be taken in order to guarantee a sustainable data distribution that uses the full capacity of the storage infrastructure: if minimum data reorganization is not enough to keep with the

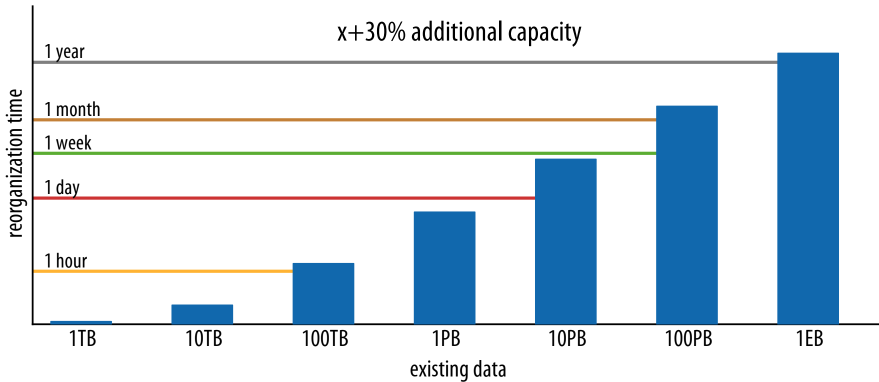


Figure 4.1: Bars depict the time spent performing data transfers due to rebalancing procedures with current technology. Each upgrade increases the capacity of the system by 30% and data transferred to new devices is the minimum required to regain a balanced load. We assume (optimistically) that the migration process can use the full capacity of the storage interconnection network and that devices cope with it. We also assume that client I/O during the reorganization process is stopped (Parameters used: Fibre Channel 32GFC–6400 MBps full duplex [9]; 30% increase/6 months [83]).

growth of Petascale and Exascale environments, we will have to design systems that migrate even less data. However, if we reorganize less than the ideal amount of data when upgrading the system, we will negatively affect load balance and performance.

One solution to this, seemingly unsolvable, paradox is to stop considering data blocks as anonymous entities and begin taking into account their semantic properties. Consider, for instance, data that is rarely (if ever) used: this kind of data does not benefit much from a balanced distribution and, thus, it makes no sense to spend significant efforts to optimize how it is laid out. Frequently accessed data, on the other hand, receives the majority of client accesses and can significantly benefit from the improved performance and parallelism of an adequately balanced distribution. Thus, this "simple" classification could significantly reduce the amount of work needed during an upgrade process, assuming that frequently used data is a small proportion of the overall dataset.

Besides, the study of the relations between data blocks is also of value. The migration process offers an opportunity to tailor where specific blocks are placed: related data blocks can be clustered in a device's neighboring regions in order to be read sequentially; or heavily accessed data blocks could be distributed among the fastest devices in order to improve their access rate and avoid potential bottlenecks.

The main contribution of this chapter is an analytical study of long-term access patterns in large-scale storage systems. We try to determine semantic properties and interrelations between data blocks, in order to establish a solid foundation on which to build better allocation strategies or prediction algorithms to improve current ones. Specifically, we focus on long-term working sets and the lifespans of data blocks, and how they evolve over time. In order to broaden the scope of the study, we apply this analysis to eight sets of traces collected from different storage systems and at different points in time, in an attempt to establish patterns common to different workloads and their historical persistence. Additionally, in the last section of the chapter we verify our observations with four extra traces that we acquired at a later time, to see if the conclusions derived from our first analysis also apply to them.

4.2 Methodology

Though there are already several studies about data access patterns and locality, most of them focus on short-term access patterns and are normally performed from a process or file perspective [13, 42, 105, 118]. This is fine for data heavily used for a few seconds, since the extremely fast transfer rates of memory allow it to be cached when needed. For storage devices, however, data transfers are significantly slower,[2] therefore any access pattern that can be exploited must involve stable working sets that survive for a long period of time or change gradually. For this reason, we focus our study on long-term block access patterns, with a minimum granularity of 24 hours, since they offer the best opportunities for improvement.

[2] *Around two orders of magnitude.*

4.2.1 Traces

In order to make the study as complete as possible, we use a set of eight different traces each representing a different computing environment. The traces used, though relatively old, correspond to well-known datasets that cover a wide range of workloads and have not been studied for long-term access locality. We study traces collected at HP Labs (cello99), the University of Harvard at Cambridge, Massachusetts (deasna and home02) and at a feature animation company (render, vcs, dbs, nfss and nfsc). Note that whenever we need to address this last set of traces as a group, we will refer to them as the animation collection. Table 4.1$_{86}$ summarizes these traces which we describe in detail in the following paragraphs:

Table 4.1: Summary of traces examined

Trace	Environment	Date	I/O level	Trace length	Read accesses	Blocks read	Avg. rd/block
cello99	research	1999	block	352 days	734,239,483	377,484,947	1.945
render	rendering	2003	NFS	25 days	1,941,929,527	170,068,738	11.418
vcs	versioning server	2003	NFS	8 days	25,799,219	9,202,048	2.804
dbs	database	2004	NFS	12 days	172,851	146,174	1.183
nfss	file server	2003	NFS	20 days	1,159,197,936	144,597,245	8.017
nfsc	caching server	2004	NFS	7 days	621,741,766	31,451,864	19.768
deasna	research & email	2002	NFS	38 days	1,356,254,187	49,156,529	27.591
home02	home share	2001	NFS	110 days	2,755,799,037	197,645,388	13.943

- The cello99 traces are a set of I/O traces that have been used in many I/O-related studies [75, 106, 150, 42]. Collected at HP Labs in 1999, cello99 capture I/O workloads from a typical research computer cluster. These traces are particularly interesting as they run for almost a year which makes them suitable for researching long-term locality patterns. Notice that trace data is missing for two days and is incomplete for nine days. Nevertheless, this does not affect the accuracy of our analysis.

- The animation traces were collected from the NFS file system of a feature animation company between fall 2003 and spring 2004. The traces were collected at several network locations, and include a pair of render racks (render), a version control server (vcs), a commercial database server (dbs), various NFS servers and an NFS cache (nfss and nfsc, respectively). In our analysis we consider each one as an individual environment to study.

  Trace data seems incomplete for two days in the render and nfss datasets. Also, even though the traces were collected daily over several weeks, the render dataset has a large gap of fifteen days just after the first three days of traces. After that, it continues normally.

- The deasna traces [42] were taken from the NFS system at Harvard's Department of Engineering and Applied Sciences over the course of 6 weeks, in mid-fall 2002. This system's workload is a mix of research and email.

- The home02 traces [42] were collected from one of the fourteen disk arrays in the Harvard CAMPUS system over 16 weeks. The CAMPUS NFS system served the majority of the school and administration at Harvard, with

over 10,000 accounts and consisted of three NFS servers, all connected to fourteen 53 GB disk arrays. Traces were collected between August 2001 and December 2001 and trace data is missing for two days.

It is worth noting that cello99 traces are block-level traces, whereas the rest of the traces are NFS level traces. This means that there should be fewer accesses to blocks for cello99 since after the first accesses, file system level caches will store block data thus reducing disk accesses. Nevertheless, this is not a problem since we only need to know whether a block was accessed or not to determine if it is shared between two different days.

4.2.2 Analysis

Due to the high amount of data involved in the study, we perform a preliminary analysis where we study data and metadata accesses separately and we try to determine an estimate of the working set overlap between different days.

Concerning metadata, this study shows significant amounts of block sharing, a result which concurs with previous studies [2, 40, 78, 118]. Nevertheless, since this behavior is already described and we did not make new findings, we will not delve further into it.

For data blocks, we must distinguish between read and write operations: while the former show important amounts of long-term sharing, the latter display very little access locality (most data blocks tend to be written and seldom accessed) which makes them less suitable for data placement optimizations. For this reason, we keep write operations out of the detailed study and focus on analyzing usage patterns for read operations, since they are more likely to exhibit long-term locality and stable working sets and, therefore, show more opportunities for improvement.

For read operations we analyze sharing proportions for individual blocks and accesses, as well as the usage history for each block in the system. Since we are particularly interested in understanding how sharing and block usage change over time, we study sharing from a temporal distance point of view, and we analyze the typical lifespan of shared blocks. We also study relevant access patterns, access sequentiality and working set density.

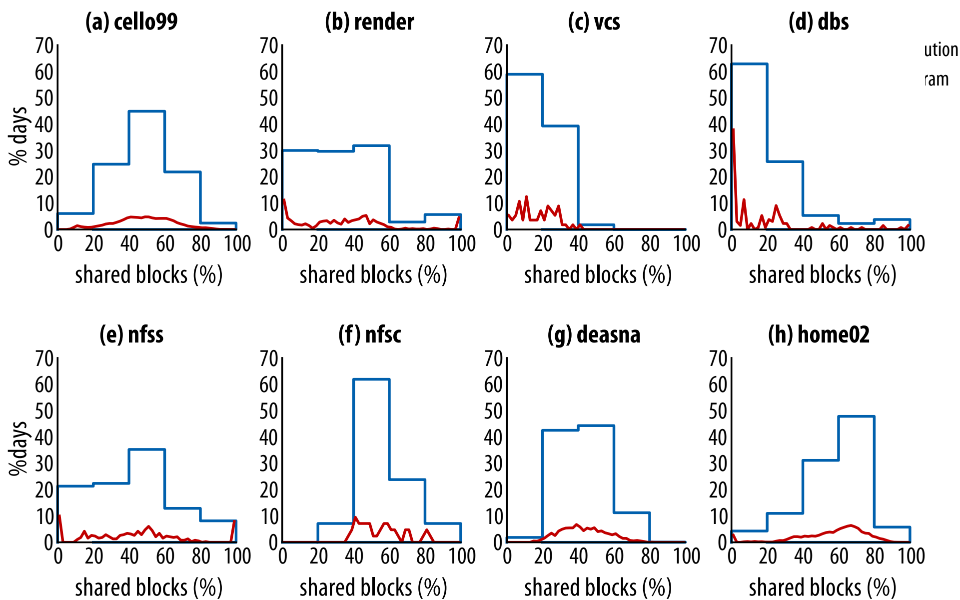


Figure 4.2: Individual and binned percentages of shared blocks by day count. The graphs in the figure depict, for each trace, the proportion *y* of examined days that showed a number of shared blocks equal to *x*. Note that we include the pure raw numbers (without binning) and also show proportions using a bin size of 20%.

4.3 Block Sharing

Histograms in Figure 4.2↑ show the distribution of shared blocks by the number of days of each dataset. For each proportion of shared blocks we plot the normalized count of days that showed that proportion. In addition, in order to provide a better perspective of the distribution of sharing across days, we aggregate the results in 20% bins and plot the number of days for each interval. For instance, for cello99 the graph shows that around 5% traced days shared *exactly* 40% of their blocks. It also shows that *in aggregate* around 45% days shared 40–60% of their blocks.

Figure 4.2↑ shows, unsurprisingly, that very different workloads have very different sharing profiles. Most days in traces cello99, deasna and home02 share between 20–80% of their blocks, with distributions mostly centered around 40–60%, 30–50% and 55–75% ranges respectively (see Figures 4.2a, 4.2g and 4.2h↑).

As expected, workloads in the animation collection vary significantly, with render, vcs and dbs rarely sharing more than 60%, 35% or 30% blocks, respectively (see Figures 4.2b to 4.2d). The distribution for nfss and nfsc, however, is more similar to that of traces cello99, deasna and home02 (see Figures 4.2e and 4.2f). This is interesting because all these environments are more interactive in nature, either because they are directly accessed by students or researchers (e.g., home02, cello99, deasna) or because they react directly to user requests (e.g., nfss, nfsc). This behavior suggests that interactive environments might have an increased sharing profile due to human repetitive access patterns.

Rendering processes, on the other hand, usually need to read a set of 3D models in order to write a scene to disk. Thus, each process reuses some file blocks (corresponding to models common to several scenes) and writes new file blocks that might or might not be used in the following days. This might explain the reduced sharing percentage when compared to interactive environments.

Note that, predictably, each dataset exhibits a large proportion of days with very few blocks shared. However, there is also a noticeable amount of days (5–10%) sharing over 80% blocks.

Observation 1 *More than half of the days studied (60%) share 40% blocks or more.*

4.3.1 Sharing and Temporal Distance

While Figure 4.2 shows the overall distribution of block sharing for each workload, it says nothing about the temporal locality of this sharing. Figures 4.390 and 4.491 show the evolution of the working set overlap in function of the distance between days: for each point (x, y) in the plot, y is computed as the *mean of the percentage of unique shared blocks* between every pair of days whose distance is x. We also compute the standard error ϵ with 95% confidence to show the variability of those percentages.[3] Note that positive distances account for the working set overlap with future days while negative distances represent the amount of overlap with past days. Figure 4.390 includes long-term mostly general purpose workloads and Figure 4.491 includes mid-term specialized workloads.

Figure 4.390 shows that an overlap of 40–60% of a day's blocks is a fairly common situation for all environments plotted, and that this overlap happens with days all over the year. It is fairly apparent, however, that there is much more overlap with future days than with past days, which might be explained by the creation of new blocks that can be shared with future days but not past days. Sharing over 60%

[3] *For instance, for trace cello99 the point $p = (100, 58 \pm 0.02)$ in the graph shows that all days at distance +100 of one another shared, on average, 58% of their blocks with 0.02% error.*

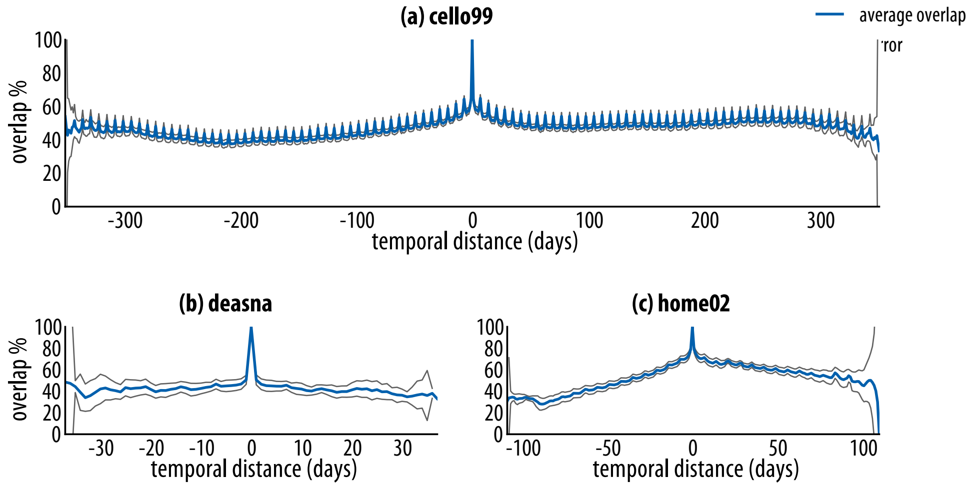


Figure 4.3: Correlation between working set overlap and temporal distance in general purpose workloads. For each point (x, y) shown in the figure, and defining $S(a, b)$ as the percentage of unique blocks common to two arbitrary days a and b, the y coordinate is calculated as the arithmetic mean $\bar{s}$ ($\pm\epsilon$ standard error) of the set composed of all $S(d_i, d_j)$, for each pair of days where i-j=x. Thus, if ϵ is small, all pairs of days at distance x had approximately $\bar{s}$ blocks in common.

of a day's blocks only happens with extremely close distances, around 15 days for cello99, 1–3 days for deasna and 25–50 days for home02 (see Figures 4.3a to 4.3c↑, respectively). This makes sense, as it is likely that blocks in use today were also used yesterday and will also be used tomorrow. An overlap above 80% is very rare.

It is worth mentioning that the three workloads exhibit a periodical pattern where the overlap is higher than normal. This can be observed in all datasets in Figure 4.3↑ as a series of peaks that are almost evenly spaced, though it is more noticeable in the cello99 trace. Careful examination of the traces shows that this increase tends to repeat every 6 or 7 days, which strongly implies that it might be related to cyclical events associated to work weeks and repetitive human behavior.

Regarding specialized workloads, Figure 4.4↷ shows, again, that workloads in the animation collection are very diverse. Workloads render, nfss and nfsc have a similar behavior to those of Figure 4.3↑ but with a lower amount of sharing: 30–40% for render, 30–50% for nfss and 40–60% for nfsc (see Figures 4.4a, 4.4d and 4.4e↷). The render workload in particular seems to favor sharing with future days which supports our hypothesis that an important proportion of newly created blocks will be reused. Workloads vcs and dbs show the least amount of over-

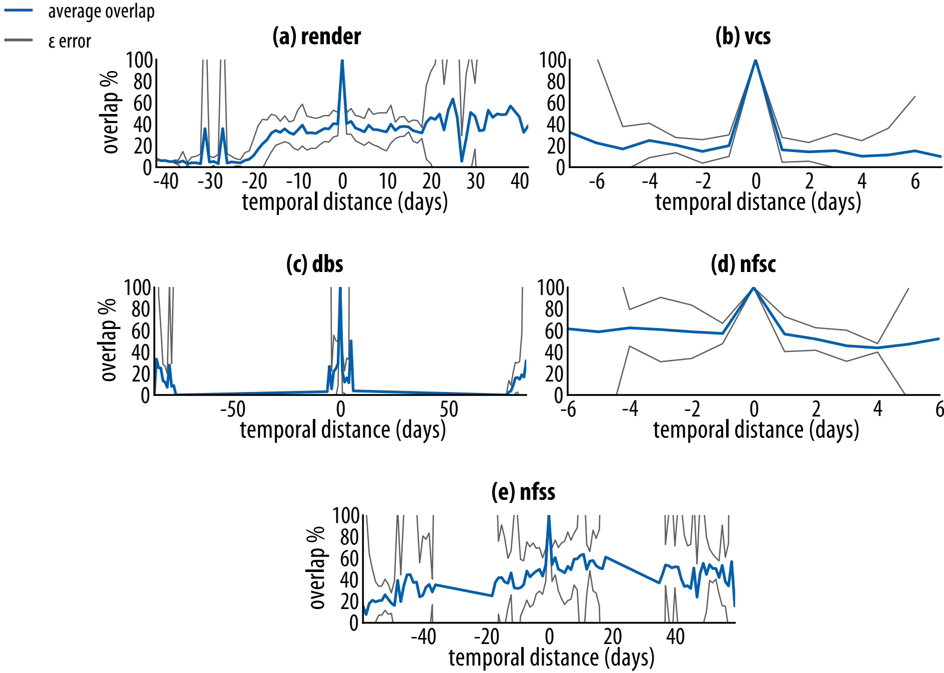


Figure 4.4: Correlation between working set overlap and temporal distance in specialized workloads. For each point (x, y) shown in the figure, and defining $S(a, b)$ as the percentage of unique blocks common to two arbitrary days a and b, the y coordinate is calculated as the arithmetic mean $\bar{s}$ ($\pm\epsilon$ standard error) of the set composed of all $S(d_i, d_j)$, for each pair of days where i-j=x. Thus, if ϵ is small, all pairs of days at distance x had approximately $\bar{s}$ blocks in common.

lap (0–10% and 10–20%, respectively) that only grows for days that are extremely close to the one examined. This is a typical behavior of source control systems, where writes (source commits) tend to dominate over reads (source checkouts), and would be expected of databases used mostly to keep information, rather than accessing it. Interestingly enough, the overall sharing for the dbs workload increases by the end of the tracing period. Lack of further data prevents us from attempting to explain this behavior (see Figures 4.4b and 4.4c$_{\uparrow}$).

Observation 2 *The overlap in working sets remains fairly stable at 30–50%, and decreases as temporal distance grows.*

Observation 3 *The overlap in working sets is heavily influenced by repetitive human behavior.*

4.3.2 Accesses to Shared Blocks

In the previous section, we saw that there is a significant temporal overlap in working sets in most workloads. However, it would be interesting to determine how often the blocks in these working sets are accessed, since shared data is of no interest if it does not receive a significant amount of activity. Histograms in Figure 4.5↷ show the distribution of accesses to shared blocks normalized by the number of days of each dataset. Like in Figure 4.2$_{88}$, we plot the normalized day count for each percentage of accesses to shared blocks, as well as the aggregate in 20% bins.

Once again, we find the duality between general purpose, mostly interactive environments and environments with specialized workloads. Accesses to shared blocks are predominant for the former, with cello99 being in the 50–70% range, deasna in the 70–95% and home02 in the 60–80% range (see Figures 4.5a, 4.5g and 4.5h↷).

Workloads in the animation collection (Figures 4.5b to 4.5f↷) are particularly interesting, because all except dbs have an important amount of days with 40–80% accesses to shared blocks: 55%, 85%, 51% and 80% for render, vcs, nfss and nfsc, respectively. Note also that render, nfss and nfsc also exhibit 10–30% days accessing over 80% shared blocks, implying that working sets for these environments might be nearly identical in those days. This would make sense for processes continuously accessing the same sets of files over several days.

Note as well that there is also a large proportion of days accessing very few shared blocks for render and nfss, with dbs being the best representative for this tendency. This suggests that these environments accessed an important amount of blocks that had not been seen before. This would make sense for short-lived, temporary files like those created as part of a rendering process. This is not conclusive, however, since there is no more data available for these environments.

Observation 4 *Shared blocks are significantly more accessed than non-shared blocks.*

4.4 Block Usage

In previous sections we have determined that there are considerable amounts of shared blocks in most of the workloads studied and that these blocks represent a large portion of daily accesses. Nevertheless, it is still unclear if this set of shared

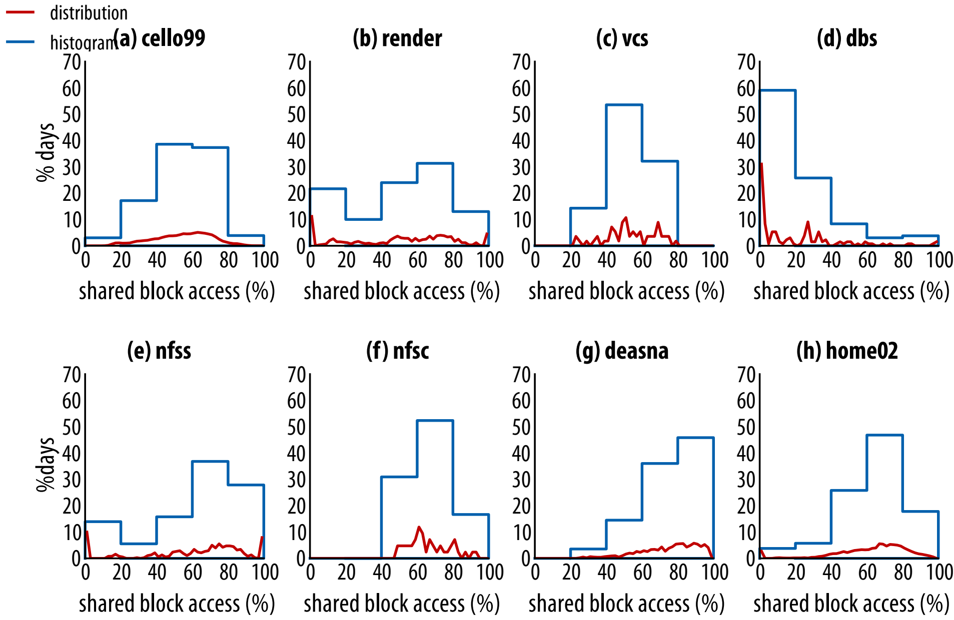


Figure 4.5: Individual and binned percentages of accesses to shared blocks by day count. The graphs in the figure depict, for each trace, the proportion *y* of examined days that showed a number of accesses to shared blocks equal to *x*. Note that we include the pure raw numbers (without binning) and also show proportions using a bin size of 20%.

blocks changes over time or the same blocks are being accessed over and over again. In this section we evaluate how long blocks "survive" and the different access patterns present in the traces.

4.4.1 Block Lifespan

In a trace, we define the "lifespan" of a block as the number of days between the first and last accesses recorded for it. Figure 4.694 depicts the cumulative distribution function (CDF)[4] of the blocks' lifespan, grouped by lifespan length for the sake of legibility. Notice that due to the existence of gaps in some traces, lifespan length can be larger than trace length.

All traces show different but relevant amounts of short-lived blocks. This is to be expected as there should be many blocks that are accessed only once or for a few days (e.g., those related to short-lived files that are created and accessed over

[4] *A CDF describes the probability that a real-value random variable X will be found at a value less than or equal to x.*

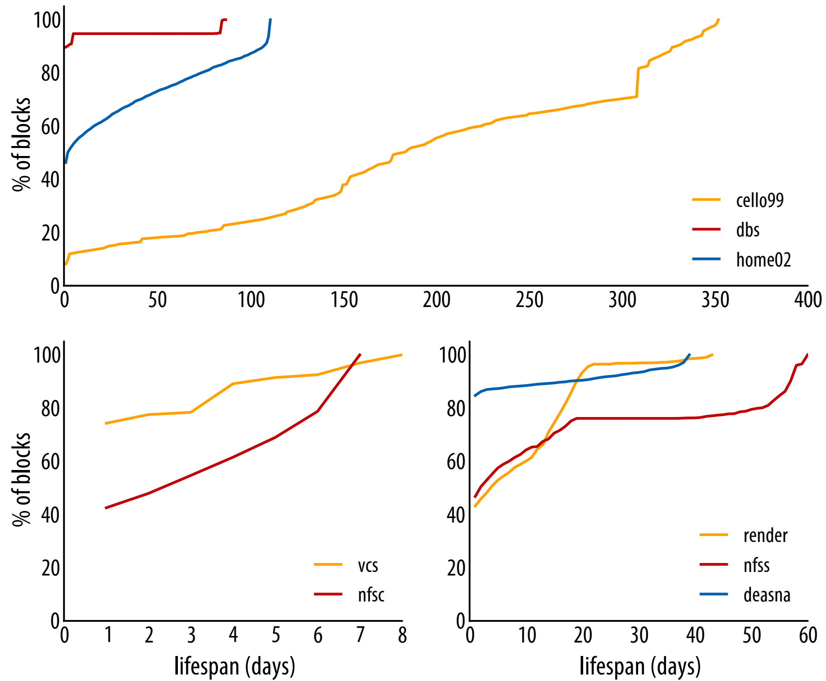


Figure 4.6: Cumulative distribution function of blocks' lifespan by block count. A coordinate (x, y) in the figure shows the percentage y of blocks with a lifespan $l \leq x$. The lifespan of a block is defined as the number of days passed between the first and last recorded accesses to it.

a week). For instance, blocks accessed in the deasna, dbs and vcs workloads are extremely short-lived, with more than 75% not being accessed again after 1 or 2 days. Workloads render, nfss, nfsc and home02 show that 40–50% blocks are not accessed for more than 4, 3, 1 and 16 days, respectively. The cello99 case stands out as short-lived blocks only represent a 13% of all examined blocks.

The relevance of medium to long-lived blocks (alive between 1–7 days) varies wildly for each environment. For the cello99, dbs, vcs, render and deasna workloads these represent but a small fraction of all blocks (between 5–8%), whereas 15–20% of all nfss and home02 blocks as well as 20–25% of nfsc blocks fall into this category.

Long-lived blocks (seen for more than a week) represent 10–12% of all blocks for deasna and dbs workloads. For render, nfss and home02 they add up to 40%, while 80% of cello99's blocks are long-lived. It is apparent that, the longer the trace, the longer the period of time where blocks are accessed repeatedly. It seems that blocks

surviving more than 1 day are very likely to live a relatively long time, which agrees with previous observations [42, 118]. Notice that we have not considered the vcs and nfsc traces due to their short duration.

Observation 5 *All workloads show important amounts of short-lived blocks (around 1-day lifespan).*

Observation 6 *Lifespans greater than 1 day follow a uniform distribution.*

4.4.2 Block Access Patterns

The previous section showed how long blocks are being used, but it said nothing about how often they are accessed during this period of time. For instance, a block only accessed once on the 1st day and once on the 200th day will have a lifespan of 200 days, but this will not reflect its real usage. Figure 4.796 shows the CDF for the actual usage of blocks in days, grouped by trace length.

We can see that although there is a lot of diversity concerning blocks' lifespan, there are evident similarities when considering the actual amount of days each block is used. Around 80% blocks in research oriented workloads like cello99 and deasna have been used for very few days (25 days for cello99 and 2–3 days for deasna), and the same happens with the dbs trace. This means that those blocks are either accessed very frequently for a short amount of time or accessed intermittently over the trace duration. Workloads vcs and render also show this behavior predominantly (around 5 days for 80% blocks).

In the case of nfss and home02 workloads, 25% blocks are accessed for more than 55 and 75 days, respectively. Given that both environments are not computation-intensive, and the inherent randomness and unpredictability of their workloads, it seems likely that this represents large amounts of data accessed periodically.

The behavior of the nfsc trace is interesting, given that it shows up to 40% blocks being accessed for more than 4 days. Given the length of the trace (7 days) and the caching behavior of this environment, this suggests clients accessing the same blocks for several days.

Observation 7 *Blocks are accessed for short periods of time, even if their lifespans are long.*

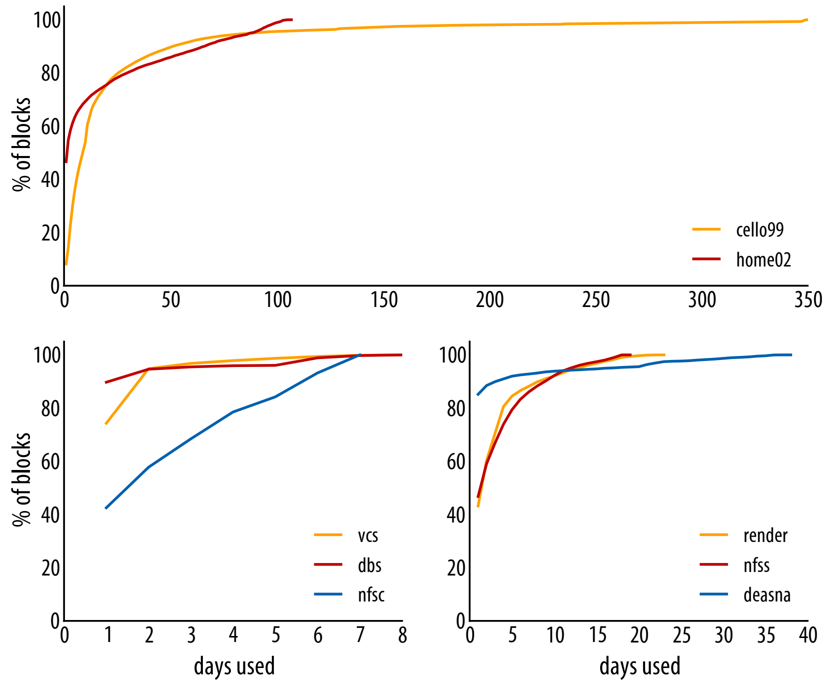


Figure 4.7: Cumulative distribution function of blocks actual usage by block count. A coordinate (x, y) in the figure shows the percentage y of blocks with an actual usage $u \leq x$. The actual usage of a block is defined as the number of days it was accessed during its lifespan.

4.4.3 Access Sequentiality

Previous analysis were useful to determine the proportions of short, medium and long-lived blocks that characterize each environment, and also whether blocks are used often during their lifetime or not. Now it would be interesting to find out whether blocks are accessed in sequential day bursts or, on the contrary, they are accessed randomly. Figure 4.8 shows the cumulative distribution of the longest periods of time when blocks are used. We consider a period of time as a series of consecutive days when a particular block is used.

It is worth noting that this notion of consecutiveness is not strict, as we have included a tolerance factor τ in order to ignore small-sized gaps. Hence, a block used during N consecutive days, that stops being used during $d \leq \tau$ days but is used again for another M consecutive days, will be considered as used $N + d + M$

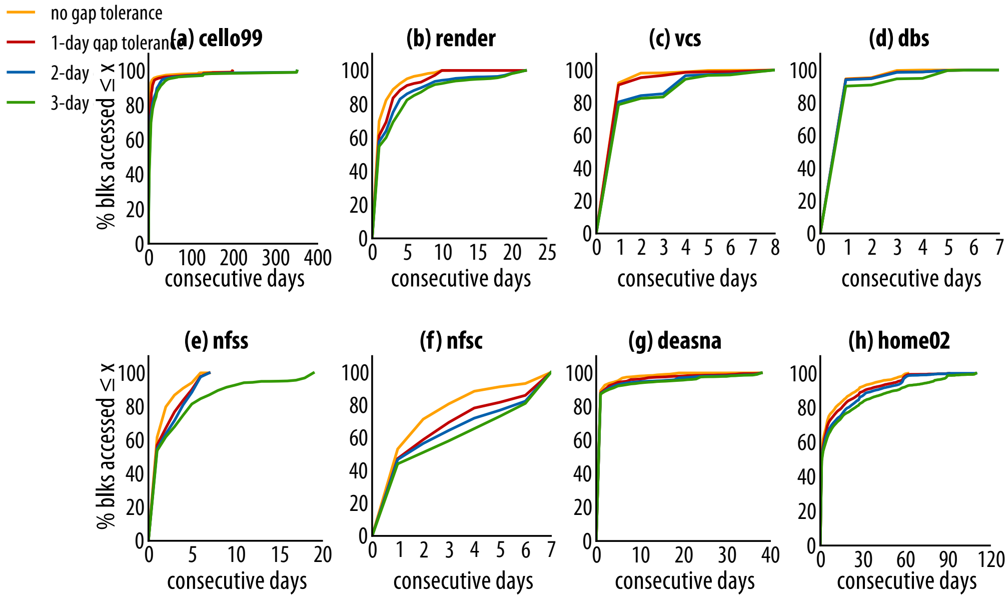


Figure 4.8: Cumulative distribution function of blocks consecutive usage by block count. A coordinate (x, y) in the figure shows the percentage y of blocks with a consecutive usage $c \leq x$. The consecutive usage of a block is defined as the number of consecutive days it was accessed, also considering a tolerance factor τ.

consecutive days. The rationale behind this decision is that, due to weekends and holidays, there might be intermittent gaps in the blocks usage. Using this method we can filter out with high probability the usage gaps due to non-working days like weekends or holidays.

Figure $4.8_{\uparrow}$ has been generated using different tolerance factors $\tau \in [0, 3]$ in order to see how they affect access sequentiality. Notice that lower tolerance factors seem to "contain" higher tolerance factors. This is to be expected since, for instance, a 3-day gap tolerance factor will ignore 3-day gaps, as well as 2-day and 1-day gaps.

When considering 0-gap tolerance, datasets in the animation collection show that about 90% of their blocks have been used consecutively for 5 days or less, though most curiously render, nfss and nfsc settle around the 5-day mark and vcs and dbs around the 1-day mark (see Figures 4.8b to $4.8f_{\uparrow}$, respectively). For general purpose environments, 90% blocks for cello99 and deasna are used consecutively for 5 days or less (Figures 4.8a and $4.8g_{\uparrow}$). Workload home02 (Figure $4.8h_{\uparrow}$) does not follow this pattern, however, with 90% blocks being accessed up to 25 consecu-

tive days. This might be explained by the different requirements between research and user workloads.

Setting τ to 1 day only increases sequentiality counts significantly for the nfsc trace (5–7 days for 90% blocks), whereas other workloads only exhibit slight increases. Nevertheless, for all workloads except dbs and nfsc, gap tolerances of 2–3 days increase sequential usage for 90% blocks by 2 and 2.5 times, respectively. Notice also that cello99, render and home02 workloads exhibit around 5% blocks with very long runs (300–350, 18–25 and 96–120 days, respectively) that are not visible with lower tolerance factors. Since the 2-day gap curves should be most "realistic" (as they include both weekends and 1-day holidays) this information could be very important when designing long-term caching algorithms.

Observation 8 *Most blocks are accessed for a few consecutive days.*

Observation 9 *Consecutive usage for blocks increases when taking into account repetitions in human behavior.*

4.4.4 Working Set Density

Finally, it would be interesting to determine if daily working sets are sparse and include a large number of different blocks, or if they are dense and a small number of blocks concentrate a large amount of accesses. Figure 4.9 plots, for each day, the number of blocks that concentrate up to 90% daily accesses, grouped again by trace length.

Figure 4.9 shows that for all environments, except cello99 and dbs, 90% accesses concentrate on 1–6% blocks with low variance. All these systems are fairly stable, occasionally showing peaks or valleys where the block count increases or decreases significantly. This is important because a careful inspection of the data contained by the animation collection shows several days where significantly less blocks are accessed. The number of blocks accessed these days is so low compared to the usual behavior for the rest of the trace, that we believe the difference might be due to tracing collection errors or anomalous situations. The case for the dbs is special since considerably more blocks are accessed at the beginning of the tracing period, thus lowering the results when compared to the rest of the trace. Nevertheless, the data provided is sensible enough to consider it as normal behavior.

General purpose environments like cello99, deasna and home02 show the variable behavior typical in interactive systems, though 90% accesses are usually made

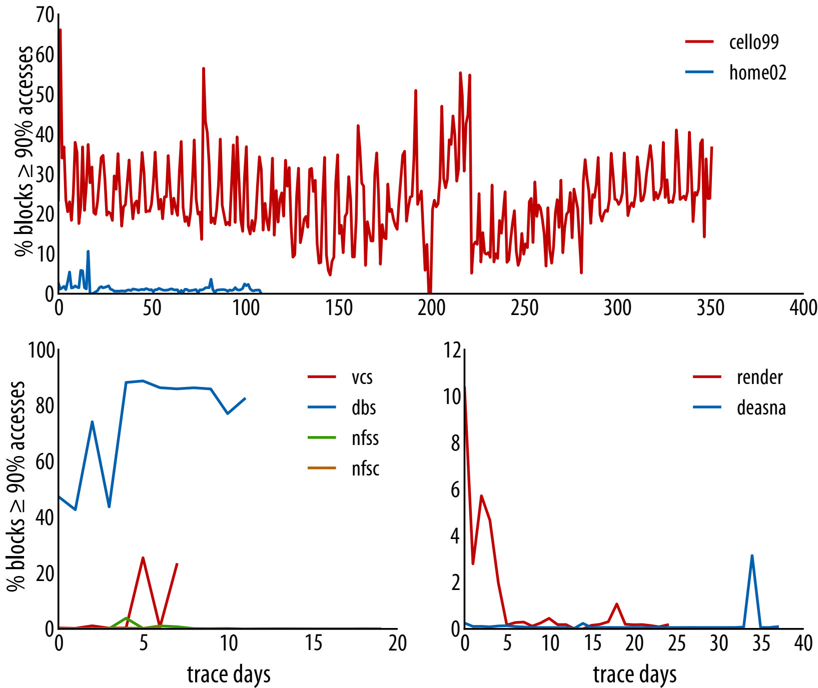


Figure 4.9: Daily count of blocks with 90% accesses. The plots in the figure show the total percentage of blocks that received more than 90% daily accesses w.r.t the trace day when they were measured.

to the same block percentages: 20–40% for cello99, 0.05–0.15% for deasna and 1–2% for home02. Notice also a peak around day 33 in deasna and a valley around day 200 in cello99. Both anomalies correspond to trace collection errors.

Observation 10 *A small percentage of blocks receives up to 90% of daily accesses.*

4.5 Scope of Our Hypothesis

In this last section, we want to verify if the hypotheses drawn from our study can be validated against additional (newer) traces, that were not included in our initial study. In order to do that, we focus on evaluating the distribution of block accesses and the working set overlap for the newer traces[5] and see if our observations in the previous sections hold. The newer traces are the MSRC and the SRCMap collections.

[5] *Our proposal for an extensible RAID strategy in Chapter 5**107* *is based on these parameters.*

Table 4.2: Summary statistics of 1-week long traces from seven different systems

Trace	Year	Workload	Reads (GB)		Writes (GB)		R/W	Accessed	Accesses to
			total	unique	total	unique	ratio	data (GB)	top 20% data
cello99	1999	research	73.73	10.52	129.91	10.92	0.62	203.65	65.77%
deasna	2002	research/email	672.40	23.32	231.57	45.45	2.54	903.97	86.88%
home02	2001	NFS share	269.29	9.07	66.35	4.49	3.94	335.64	61.36%
webresearch	2009	web server	–	–	3.37	0.51	–	3.37	51.33%
webusers	2009	web server	1.16	0.45	6.85	0.50	0.09	8.01	56.17%
wdev	2007	test server	2.76	0.20	8.77	0.42	0.21	11.54	72.44%
proj	2007	file server	2152.74	1238.86	367.05	168.88	7.33	2519.79	57.64%

The MSRC collection [101] is a set of block-level traces collected over one week at Microsoft Research Cambridge's data center in 2007. The traces include I/O requests on 36 storage volumes containing 179 disks on 13 servers. For this study, we focus on the wdev and proj servers, a test web server (4 volumes) and a server of project files (5 volumes), respectively, since they contain the workloads involving more requests.

The SRCMap collection [138] is also a set of block-level traces, but this time collected by the Systems Research Laboratory (SyLab) at Florida International University. The traces were collected for three weeks at three production systems that included, among others, an email server, an NFS file server and a virtual machine monitor. Of all the workloads traced, we focus on the webresearch and webusers workloads: the former corresponds to an Apache web server managing FIU research projects, while the latter comes from a web server hosting faculty, staff, and graduate student web sites. Once again, we choose these workloads over the others available because they involve a large number of requests.

In these new experiments, we use some of the traces from the previous sections in order to have a meaningful comparison background, and see how results differ between the newer and the older traces. Specifically, we revisit the cello99, deasna and home02 traces because they exhibited the best results concerning long-term block sharing, and we include write requests to enlarge the scope of the experiments. In all experiments, we use an entire continuous week of traced requests chosen at random rather than the whole trace, since we are only interested in validating our previous conclusions.

Figure 4.10 shows the CDF for block access frequency for each workload. As expected, all workloads show highly skewed distributions of frequency of access:

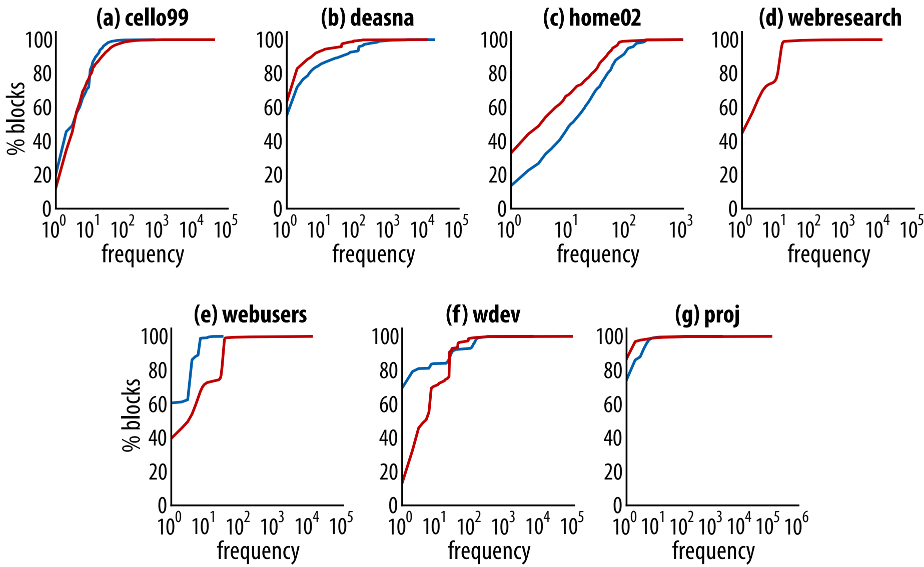


Figure 4.10: Block-frequency in the examined 1-week traces. Plots depict the CDF of block accesses for different frequencies: a point (f, p_1) on the block percentage curve indicates that p_1% of total number of blocks were accessed at most f times.

for read requests ≈76–98% blocks are accessed 50 times or less, while for write requests this value rises to ≈89–98% blocks. On the other hand, a small fraction of blocks (≈0.05–0.7%) are very heavily accessed in all cases (read or write requests).

This skew can also be observed in the measurements collected in Table 4.2↶: the top 20% most frequently accessed data blocks contribute to a significantly large fraction (≈51–83%) of all I/O, which are similar results to those reported by previous studies on the subject [51, 78, 16, 138, 95].

Thus, the following remark holds true for all the workloads examined: *data usage is highly skewed with a small percentage of blocks being heavily accessed*, which is in consonance with our earlier observations in the chapter.

Plots in Figure 4.11$_{102}$, depict the changes in the daily working-sets for each of the workloads, i.e., each bar in the figure represents the percentage of unique blocks that are accessed both in day d and day $d+1$. In all workloads there is a significant overlap (≈20–80%) in the data blocks accessed in immediately successive days. Most importantly, we observe that there is a substantial overlap even when

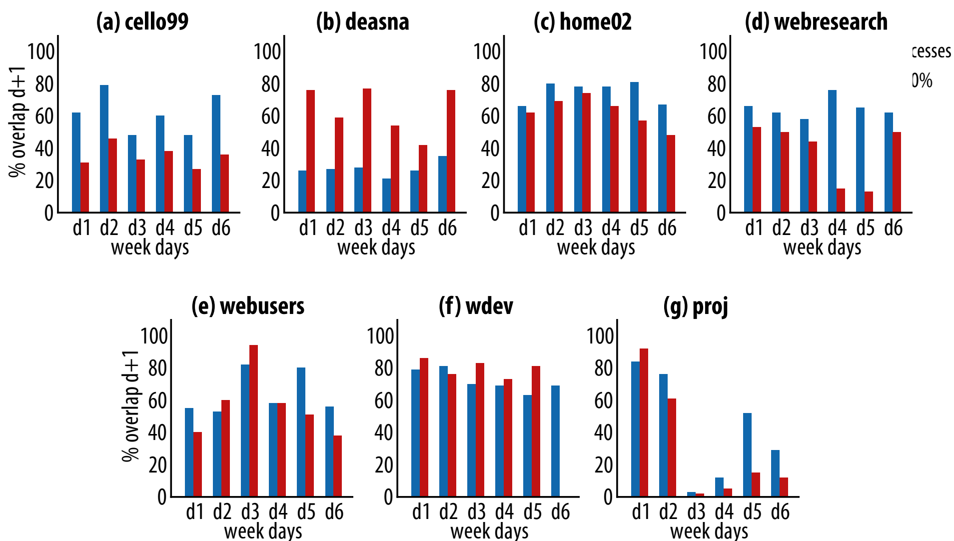


Figure 4.11: Working-set overlap in the examined 1-week traces. Plots depict the changes in the daily working-sets of the workloads: a bar (d, p_2) indicates that days d and $d+1$ had p_2% blocks in common. This is shown for all blocks and for the 20% blocks receiving more accesses.

considering the top 20% most accessed blocks. The *deasna* workload is particularly interesting because it exhibits low values of overlap (≈20–35%) when considering all accesses, which greatly improves when considering blocks in the top 20%. This implies that the working-set for this particular workload is more diverse but still contains a significant amount of heavily reused blocks. Nevertheless, this proves that *working-sets remain fairly stable over long durations*, which is one of the observations derived from the previous traces.

Based on the above remarks, it seems reasonable to exploit long-term temporal locality and non-uniform access frequency distribution to improve the extensibility of a storage system. In the following chapter, we will use this information to design a RAID extension mechanism based on redistributing copies of heavily used data blocks.

4.6 Conclusions

In this chapter we have shown an analysis of eight different network file system traces, three of which were collected over periods of time longer than a month. The traces selected include two research workloads, an NFS home share workload, a rendering workload, three server workloads and a database workload. By analyzing the daily usage of blocks and read access patterns for each environment we have found significant similarities between those with direct human influence, even though they were collected in different years. Besides, we verified the validity of our conclusions by examining the working set overlap and frequency of accesses of four, more modern, additional traces that we acquired at a later time.

The analysis of variation of block sharing across file systems demonstrates that in general purpose workloads at least 50% of the blocks used in a day are reused some other day. Specialized workloads vary depending on the work performed, though there is a noticeably high amount of block sharing in those systems where human interaction was normal.

The study of the correlation between block sharing and the temporal distance between days demonstrate similarities in systems with direct and indirect human interaction. Unsurprisingly, block sharing is higher for closer days and tends to propagate to the future rather than the past, decreasing as the distance between days increases. Interestingly, for general purpose environments there is an important amount of sharing even for extremely distant days. Specialized workloads without human interaction, on the other hand, are likely to have less sharing. A most interesting result is that week periods have a strong influence in the working set overlap. A surprising but important result is that more than half daily accesses are directed to shared blocks for all environments.

Block lifespan is very variable for different environments though similarly to previous results [42, 118] we see that blocks living longer than a day are very likely to live a relatively long time, for the interactive long-term traces studied. Furthermore, in all environments blocks are more likely to be accessed over a few consecutive days. Finally, the most important result of this analysis is the proof that 90% daily accesses are directed to a small subset of blocks.

In summary, the analysis demonstrates that most environments have a high amount of blocks shared over time, that most accesses are made to shared blocks, and that the majority of those accesses is concentrated in very few blocks.

In the next chapter we will see how these findings can be used to design a prediction-based, long-term caching strategy that determines future working sets with high probability, which allows it to improve the extensibility and performance of a RAID architecture.

Chapter 5

Extensibility in RAID Architectures

"We can only see a short distance ahead,
but we can see plenty there that needs to be done."
— Alan M. Turing

5.1 Motivation

Randomized data layouts yield a global uniform block distribution across all storage systems, support heterogeneous environments and provide minimal data reorganization when upgrading the storage system. Does this mean that they are a perfect solution for all problems? Not by a long shot.

First of all, in the introduction of Chapter 483 we already make a good point of why minimal data reorganization is not sufficient in Petascale and Exascale storage environments: it is simply too much information to move efficiently! Additionally, however, file systems typically try to allocate together a file's blocks [88], in order to use spatial and temporal locality to improve access performance, which can be problematic for randomized distributions that scatter blocks across the data space, even if it is in a controlled manner. Though this can be (somewhat) alleviated by increasing the block size, which helps maintain spatial locality, it can cause problems to applications that rely on fine-tuning the allocation unit[1] to improve their I/O throughput [30].

[1] *Scientific and database applications commonly use data block sizes between 8 KB and 256 KB, due to their high concurrency and small request size.*

(a) Original RAID-0

disk 0	disk 1	disk 2	disk 3	disk 4
0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24
25	26	27	28	29

(b) Upgraded RAID-0

disk 0	disk 1	disk 2	disk 3	disk 4	disk 5	disk 7	disk 8
0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29		

Figure 5.1: RAID restriping process. The figure shows the layout of data blocks in a RAID-0 array before and after being upgraded using a restriping process. This process simply rebuilds the Round-robin distribution of data blocks extending it to the new stripe size. Note that 25 out of 30 blocks ($\approx$83.4%) migrate to different locations in the disk. Numbers inside each cell represent the logical block identifier.

On the other hand, storage architectures based on RAID are still a popular choice to deploy reliable and high performing storage into environments with intensive I/O requirements, with acceptable economic and spatial costs. In this kind of architectures, capacity and performance upgrades are usually achieved by adding new devices to the existing RAID array and *restriping* (reorganizing) the current data set. Extending RAID architectures in this manner, however, poses several new challenges not present in randomized data distributions:

1. To regain the uniformity of the data distribution, certain blocks must be moved to the newly added disks. The traditional approaches that try to preserve the Round-robin order [25, 52, 153] end up redistributing large amounts of data blocks between old and new disks, regardless of what the numbers of new and old disks are (see Figure $5.1_{\uparrow}$).

2. Alternative methods that migrate only a minimum amount of data, can have problems to keep a uniform data distribution after several upgrade operations (like the Semi-RR algorithm [50]), or end up producing a data layout that does not use all the available hardware at all times (GSR [151]), thus limitting the array's performance.

3. Some existing RAID solutions like RAID-5 and RAID-6 offer redundancy mechanisms by computing erasure codes or parities that allow to reconstruct data blocks if one or several devices fail. Extending this kind of strategies can be

problematic due to the additional overhead of recalculating and updating the associated parities, as well as the necessary metadata updates associated with stripe migration.

4. Furthermore, RAID solutions are widely used in online services where clients and applications need to access data constantly. In these services, the downtime cost can be extremely high [110], and thus any strategy to upgrade RAID arrays should be able to interleave its job with normal I/O operations.

For all these reasons, in this chapter we focus on developing a mechanism that can be used to extend RAID arrays in a Petascale or Exascale architecture, while solving the challenges we just mentioned. The main contribution of this chapter, thus, is an extension of the standard RAID layout called CRAID, which minimizes the overhead of the upgrade process by redistributing "relevant data" in real-time. To do that, CRAID[2] tracks data that is currently being used by clients and reorganizes it in a specific, dedicated partition. This allows the disk array to maintain the performance and distribution uniformity of the data that is actually being used by clients and, at the same time, significantly reduces the amount of data that must be migrated when new devices are installed.

[2] *Which, by the way, is pronounced as [cee·reid], with a short 'i'.*

The design of this RAID extension is based on the notion that providing good levels of performance and load balance for the current working set suffices to preserve the QoS[3] of the RAID array. This idea is born from three key observations about storage workload characteristics, which are distilled from our earlier discussion in Chapter 4_{83}:

[3] *Note that, throughout this chapter, the term QoS refers to the performance levels and load distribution quality offered by a RAID array and is unrelated to SLAs.*

1. Data in a storage system displays a *non-uniform access frequency distribution*: when considering coarse-granularity time spans, "frequently accessed" data is usually a small fraction of total data.

2. This active data set exhibits *long-term temporal locality* and is *stable*, with small amounts of data losing or gaining importance gradually.

3. Even within the active working set, data usage is heavily skewed, with "really popular" data *receiving over 90% of user accesses*.

Though these observations are largely intuitive and in consonance with the findings of other researchers [51, 62, 121, 5, 120, 140, 138, 16], to our knowledge there have not been any attempts to apply this information to the extension of large-scale RAID architectures.

5.2 Proposal: CRAID

In this section we describe CRAID, our proposal for a self-optimizing RAID array that performs an online block reorganization of frequently used, long-term accessed data, in order to reduce the overhead of large-scale rebalancing. Two are the main ideas behind the design of CRAID: first, we intend to prove that it is possible to efficiently extend deterministic, rule-based data distributions for large-scale storage with good performance and load balance; second, we want to demonstrate that the layout used for "cold" data (i.e., data that is not currently being accessed) is not very important as long as I/O access to currently important data is heavily optimized. We will see in the following section whether these two ideas can be held true in a realistic setting or not.

5.2.1 Description

The goal that drives CRAID is to reduce the amount of data that needs to be migrated during reconfigurations while providing QoS levels similar to those of traditional RAID arrays. In order to achieve this, CRAID claims a small portion of each device and uses it to create a *cache partition* (P_C) that will be used to place copies of heavily accessed data blocks. The aim of this partition is to separate data that is currently important for clients from data that is rarely (if ever) used. Data not currently being accessed is kept in an *archive partition* (P_A) that uses the remainder of the disks. Note that this partition can be managed by any data allocation strategy, but it is important that it can grow gracefully (though without the strict performance requirements of "hot", live data) and that any archived data can be accessed with acceptable QoS, so as not to hinder data transfers between partitions (see an example of a RAID-0+RAID-0 CRAID layout in Figure 5.2112).

Using two independent partitions for hot and cold data allows for a more appropriate management of these two data sets, and also offers the opportunity to apply layout optimizations suited to each of their requirements. In particular, effectively optimizing the layout of data within a small confined partition offers several potential benefits:

1. It is possible to create a large cache by using only a small fraction of all available disks, which allows important data to be cache-resident for longer periods of time with a small capacity tradeoff.

2. A disk-based cache is a persistent cache: this means that any optimized layout continues to be valid as long as it is warranted by access semantics, even if it is necessary to shutdown or reconfigure the storage system.

3. The size of the partition can be easily configured by an administrator or an automatic process to better suit storage demands.

4. Clustering frequently accessed data together offers the opportunity to improve access patterns: data accesses that were originally scattered can be sequentialized if the layout is appropriate. This also helps reduce seek times and rotational delays in all disks since hot data blocks are placed close to each other.

5. As we have seen, whenever new devices are added, the current strategies need to redistribute large amounts of data in order to be able to use the hardware effectively and maintain QoS levels (e.g. performance or load balance). A disk-based cache offers a unique possibility to maintain QoS by redistributing only the most accessed data. This significantly reduces the cost of the extension process, since it limits the amount of data being worked on.

6. A partition extended over several devices has two advantages over using dedicated disks. First, it maximizes the potential parallelism offered by the storage system. Second, it is much more likely to saturate a reduced set of dedicated disks than a large array. Third, benefits can be gained with the existing set of devices, without having to acquire more.

Figure 5.3113 shows the control flow supported by CRAID's architecture and its main components. When an I/O request enters the system (see control path A in the figure), it is captured by CRAID's I/O Monitor which determines if the data accessed must be considered "active". If so, data blocks are copied to the caching partition if they are not already in it and an appropriate mapping $[LBA_{orig} \mapsto LBA_{cache}]$ is stored in the Mapping Cache (paths B.1 and B.2). From this point on, an I/O Redirector will redirect all future accesses to LBA_{orig} to the caching partition (paths C.1 and C.2). This continues until the I/O Monitor decides that a block is no longer active and removes the entry from the Mapping Cache. Any update to the block's contents is then written back to P_A (path D).

Hence, the upgrade process begins immediately when a new disk is added to CRAID, (which forces P_C to grow), and is interwoven with the array's normal I/O

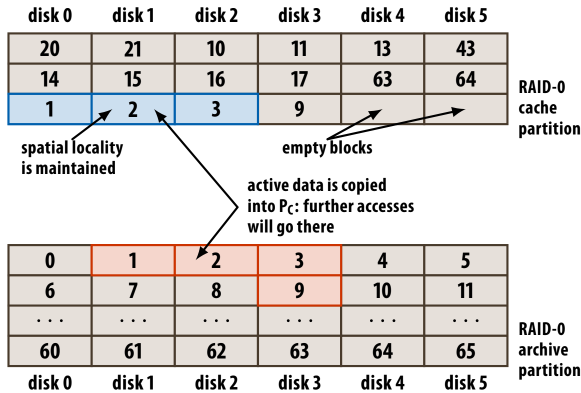


Figure 5.2: Concept of a CRAID architecture with RAID-0 in both partitions. The caching partition is assembled with segments from all five disks. Active data is then copied into it, where it is accessed and updated. When a data block is no longer active, the original block in the archival partition is updated and the copy is removed from the caching partition.

operation. This permits CRAID to use the new disks from the moment they are added to the array.

In order to better understand CRAID's inner workings, in the following sections we elaborate on its design details by discussing each of the individual components mentioned in the previous control flow: the I/O Monitor, the Mapping Cache, and the I/O Redirector.

5.2.2 I/O Monitor

The I/O Monitor is the "intelligent" component responsible for analyzing I/O to identify the current working set, and schedule the appropriate operations to copy data between partitions. The I/O Monitor uses a conservative definition of working set that includes the latest k distinct blocks that have been *more active*, where k is P_C's current capacity.

Whenever a request requires a block copy that exceeds P_C's current capacity, the I/O Monitor checks if the cached copy is dirty and, if so, schedules the corresponding I/O operations to update the original data. Otherwise, the datum is replaced by the newly cached block, since there is no need to synchronize its contents back to P_A. Currently, the I/O monitor supports the following simple policies in order to make replacement decisions:

- *Least Recently Used (LRU)* uses the recency of access to decide if a block has to be replaced: the algorithm simply discards the oldest block within the working set.

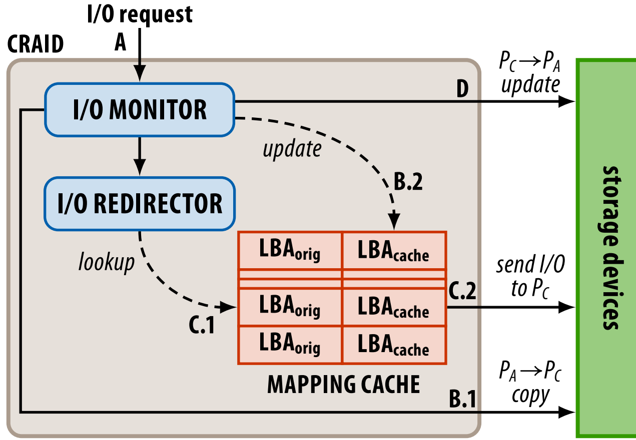


Figure 5.3: Main software components and I/O control flow of a CRAID controller. The I/O Monitor captures incoming data requests and analyzes them to determine the current set of hot data blocks. If a block is considered hot, the I/O Monitor schedules the appropriate I/Os to copy it into the cache partition, and updates the Mapping Cache accordingly. After that, subsequent requests to hot blocks are forwarded to the approppriate partition by the I/O Redirector, that uses the Mapping Cache to determine the block's specific location.

- *Least Frequently Used with Dynamic Aging (LFUDA)* uses popularity of access and replaces the block with the smallest key K_i such that:

$$K_i = (C_i \times F_i) + L, \tag{5.1}$$

  where C_i is the retrieval cost, F_i is a frequency count and L is a running age factor that starts at 0 and is updated for each replaced block [7].

- *Greedy-Dual-Size with Frequency (GDSF)* is similar to LFUDA, but includes the size of the original request, S_i, to scale the weight of the blocks. The GDSF strategy [68, 28, 7] replaces the block with minimum key K_i such that:

$$K_i = (C_i \times F_i)/S_i + L \tag{5.2}$$

- *Adaptive Replacement Cache (ARC)* balances between recency and frequency of access in an online and self-tuning fashion. ARC adapts to changes in the workload by tracking *ghost hits* (i.e., recently evicted cache entries) and re-

places either the least recently used block or the least frequently used block, depending on recent history [89].

- *Weighted LRU (WLRU)* is a simple extension of the LRU algorithm that tries to find the least recently used block that is also *clean* (i.e. not dirty). In order to avoid lengthy $O(k)$ traversals it uses a parameter $\omega \in \mathbb{R}$ to limit the number of blocks that will be evaluated to $k \cdot \omega$. This also helps reduce the number of I/O operations needed to keep consistency: if the data block replaced has not been modified, there is no need to copy it back to P_A. If no suitable candidate is found, the least recently used block is replaced.

We evaluate the effectiveness of these basic strategies to accurately predict variations in the workload in Section 5.4 (p. 121). We implemented these basic strategies into our prototype, instead of more complex ones, because these algorithms are typically extremely efficient and consume few resources, which makes them suitable to be included in a RAID controller. Furthermore, their prediction rates are usually quite high. Improving the I/O monitor by exploring more complex strategies and data mining approaches is one aspect of our future work.

The I/O monitor is also in charge of rebalancing P_C. When new devices are added, the I/O monitor invalidates all the blocks contained in P_C (writing back to P_A the copies that need updating) and starts filling it with the current working set when blocks are requested. This conservative approach allows to create long sequential chains of potentially related blocks, which improves the sequentiality and parallelism of the data in P_C. Note that since P_C always holds 'hot blocks', the rebalancing is never completely finished unless the working set remains stable for a long time. Nevertheless, as we show in the following sections, the cost of this 'on-line' rebalancing is amortized by the performance obtained.

5.2.3 Mapping Cache

The Mapping Cache is an in-memory data structure used to translate block addresses in the P_A to their corresponding copies in P_C. The structure stores, for each block copied to P_C, the block's LBA in P_A, the corresponding LBA in P_C and a flag indicating whether the cached copy has been modified or not.

Our current implementation uses a tree-based binary structure to handle mappings, which ensures that the total time complexity for a lookup operation is given by $O(log\ k)$. Concerning memory, for every block in P_C, CRAID stores 4 bytes for

each LBA and 1 dirty bit, plus 8 additional bytes for the structure pointer. Assuming that all k blocks are occupied, that the configured block size is 4 KB and a P_C size of S GB, the worst case memory requirement is $2 \times S$ MB for LBAs, $S/2^5$ MB for the dirty information, and $4 \times S$ MB for the tree pointers. Thus, in the worst case, CRAID requires memory of 0.58%[4] the size of the cache partition, or approximately 5.9 MB per GB, an acceptable requirement for a RAID controller.

[4] *That is,* $(6S + S/2^5) \times 100/1024S$.

Notice that the destruction of the Mapping Cache can lead to data loss since block updates are performed in place in the cache partition. To avoid this, failure resilience is provided by maintaining a persistent log of which cached data blocks have been modified and their translations. This ensures that these blocks, whose cached copies were not identical to the original data, can be successfully recovered. Blocks that were not dirty in P_C do not need to be recovered and are invalidated.

5.2.4 I/O Redirector

The I/O Redirector is responsible for intercepting all read and write requests sent to the CRAID volume and redirect them to the appropriate partition. For each request, it first checks the Mapping Cache for an existing cached copy. If none is found, the request is served from P_A. Otherwise, the request is dispatched to the appropriate location in P_C. Multi-block I/Os are split as required.

5.3 Methodology

In order to evaluate our proposal under realistic conditions, we use detailed simulations where we replay some of the real-time storage traces described in Chapter 483. Note that some of these traces include data collected over several weeks or months, which makes them intractable for fine-grained simulations. For this reason, we simulate an entire continuous week[5] chosen at random from each dataset. Specifically, the traces chosen to run the simulations are cello99, deasna, home02, webresearch, webusers, wdev and proj. Note that we discard the animation collection in order to keep the number of experiments contained.

[5] *Just about 168 hours.*

The first set of experiments (which we review in Section 5.4121), evaluates the ability of the management algorithms presented in Section 5.2.2112 to effectively predict and adapt to changes in the working set. The second set, on the other hand, is designed to measure the performance and workload distribution of CRAID under

conditions as realistic as possible. Since CRAID's goal is to extend current RAID architectures like RAID-5 and RAID-0, this second set of experiments is further divided into two independent groups of comparative simulations: one for CRAID configurations based on RAID-5 and another for configurations based on RAID-0.

5.3.1 Configurations based on RAID-5

This first group of simulations evaluates how CRAID configurations based on RAID-5 perform against pure RAID-5 distributions. This allows us to see how the overhead of computing, accessing and writing parities affects CRAID, and we can also compare how it performs against current RAID-5 deployments. These simulations use the following data configurations:

- **RAID-5:** A RAID-5 configuration that uses all disks available. Stripes are as long as possible but are divided into *parity groups* to improve the robustness and recoverability of the array (Figure 5.4a$_{118}$). Results for this data configuration serve as an *ideal baseline* as it provides maximum parallelism and ideal workload distribution. Notice, however, that extending such an array in real life can be prohibitively expensive.

- **RAID-5^{+}:** A RAID-5 configuration that has been expanded and restriped several times. Each expansion phase adds 30% additional disks [83] that constitute a new independent RAID-5. Thus, the system can be considered a collection of independent RAID-5 arrays (or *Logical Volumes*), each with its own stripe size, that have been united to expand the storage capacity (see Figure 5.4b$_{118}$). Results for this configuration establish a *real baseline* for a more realistic storage system that has been upgraded many times.

- **CRAID-5:** A CRAID configuration that uses RAID-5 both in the cache partition and the archive partition (see Figure 5.4c$_{118}$). This strategy lets us evaluate the effect of transferring data from and to the cache when the archive partition is ideally distributed.

- **CRAID-5^{+}:** A CRAID configuration that uses RAID-5 for the caching partition and RAID-5^{+} for the archive partition (see Figure 5.4d$_{118}$). This lets us evaluate the impact of our hybrid architecture in a more realistic storage system, where the archive partition grows as a collection of Independent Logical Volumes, and the cache partition as a unified data space.

- **CRAID-5$_d$** and **CRAID-5$_d^+$:** CRAID configurations that are analogous to CRAID-5 and CRAID-5$^+$ but using a fixed number of dedicated disks for the cache partition (see Figures 5.4e and 5.4f$_{118}$). This allows us to evaluate the advantages, if any, of using dedicated hardware instead of a partition that grows with the number of disks.

- **RandSl$_p$:** Additionally, we also evaluate a Random Slicing configuration with support for parity groups as we are interested in seeing how RAID-5-based CRAID fares against our previous proposal. Nevertheless, the default implementation of Random Slicing offers fault tolerance via replication, which cannot be meaningfully compared against parity methods like those used in RAID-5. For this reason, we extend the Random Slicing implementation with a parity mechanism so that the comparison makes sense.

Please note that in the following sections we may need to refer to the CRAID configurations explained above as a group. Whenever this happens we will use the term CRAID-5* to include all the different variations.

5.3.2 Configurations based on RAID-0

Similarly to the first group, the second group simulates several CRAID configurations based on RAID-0 and compares them against a pure RAID-0. This group of experiments allows us to determine the effect of the caching partition in a pure parallel distribution like RAID-0. These simulations use the following data layouts, an overview of which can be seen in Figure 5.5$_{119}$:

- **RAID-0:** A RAID-0 configuration that uses all disks available. Stripes are as long as possible with no parity groups or other fault tolerance mechanism. Results for this layout serve as an *ideal baseline* as it provides maximum parallelism and ideal workload distribution. Similarly to RAID-5, however, extending such an array in real life can be prohibitively expensive.

- **SEQ:** This strategy simply places data sequentially in a device. When the device in question is full, SEQ chooses the next device in the array and proceeds to fill it sequentially as well. Obviously, this kind of strategy offers very poor results regarding performance (a request can only be served by one or two devices at most) and load balance (data fills up disks one at a time, leaving the others unused), but serves to determine how CRAID can improve its results in a combined strategy.

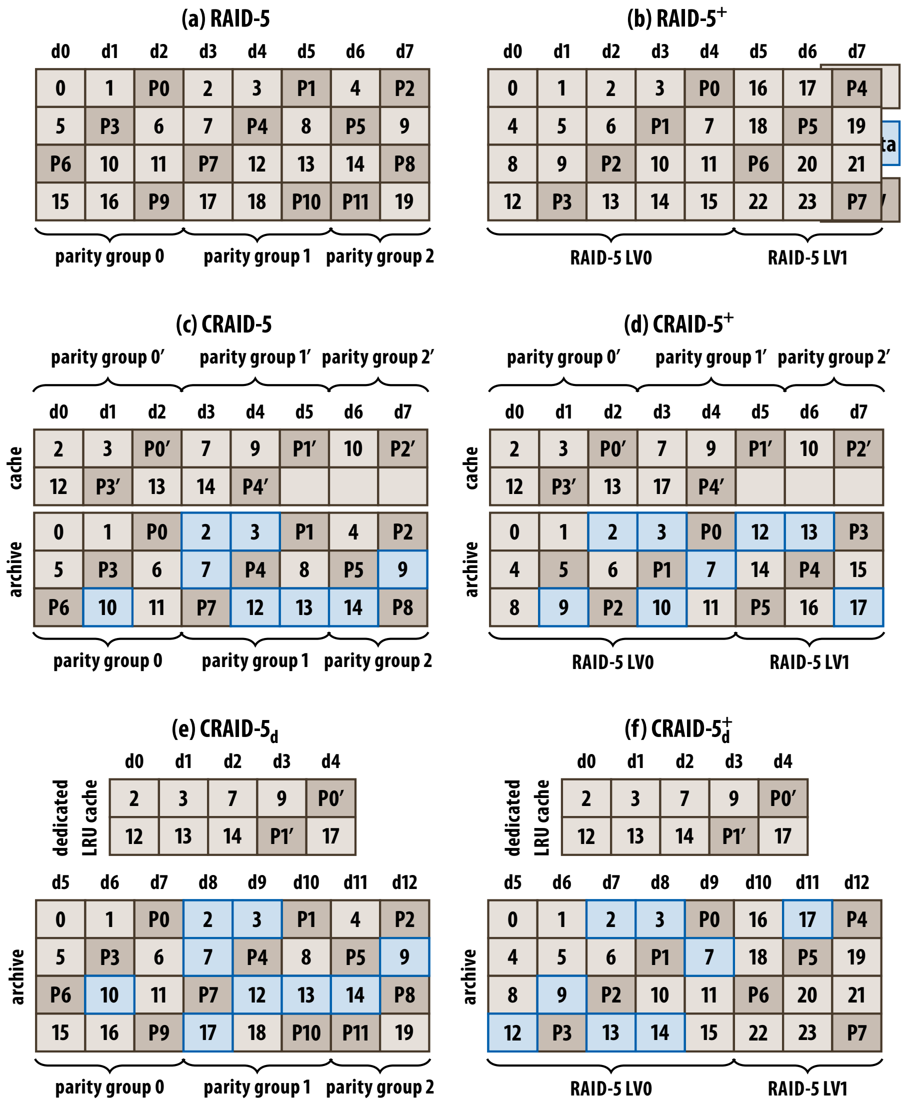


Figure 5.4: Overview of the different configurations based on RAID-5 that are evaluated in the first group of simulations. Note that in all configurations based on RAID-5 stripes use all data disks, whereas in configurations based on RAID-5_d^+ stripes in the archive partition are confined to independent logical volumes.

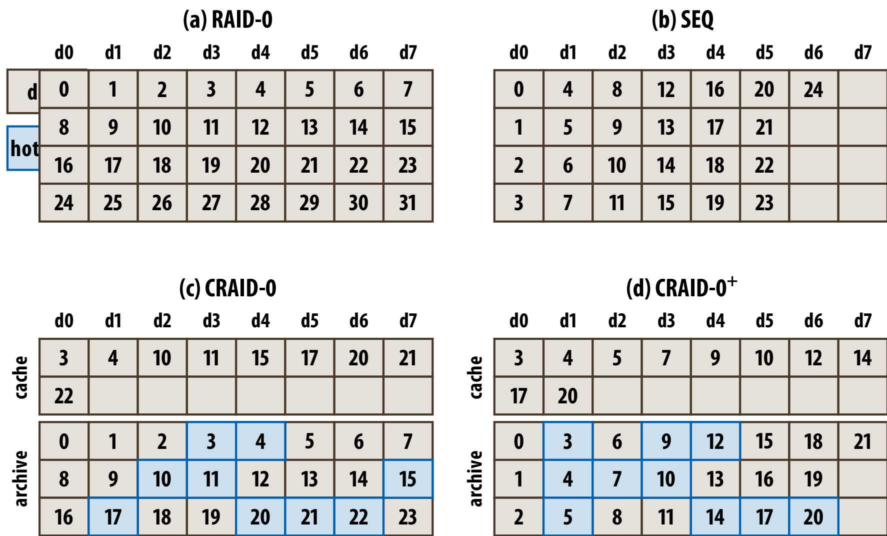


Figure 5.5: Overview of the different configurations based on RAID-0 that are evaluated in the second group of simulations. Note that in all configurations based on RAID-0 data is striped across all available disks, whereas in configurations based on SEQ, data blocks in the archive partition are allocated sequentially as long as disk capacity is not full.

- **CRAID-0:** A configuration that uses RAID-0 both in the cache partition and archive partition. This strategy lets us evaluate if a dedicated RAID-0 distribution for frequently used data outperforms a global RAID-0 layout.

- **CRAID-0$^+$:** A CRAID configuration that uses RAID-0 for the cache partition and SEQ for the archive partition. With this combination we intend to examine the effect of CRAID in a bad allocation policy like SEQ.

- **RandSl:** Additionally, we also evaluate a Random Slicing configuration since we are interested in seeing how RAID-0-based CRAID fares against our previous proposal.

Please note that, like before, in the following sections we may need to refer to the CRAID configurations explained above as a group. Whenever this happens we will use the term CRAID-0* to include all the different variations.

5.3.3 Simulation system and parameters

The simulator itself consists of a Workload Generator and a simulated storage subsystem, which is composed of an array controller and the appropriate storage components. For each request recorded in the trace files, the workload generator issues a corresponding I/O at the appropriate time and sends it to the array controller.

The array controller's main component is the I/O Processor which encompasses the functions of both the I/O Monitor and the I/O Redirector. According to the incoming I/O address it checks the Mapping Cache and forwards it to the segment of the caching partition of the appropriate disk.

The Workload Generator, the Mapping Cache and the I/O Processor are implemented in C++, while the different storage components are implemented by a disk simulation tool called DiskSim. DiskSim [26] is an accurate and thoroughly validated disk system simulator developed by the Carnegie Mellon University. It has been used extensively in several research projects for the study of storage system architectures [3, 102, 154, 80].

All experiments use a simulated testbed consisting of several Seagate Cheetah® 15,000 rpm disks [128], each with a capacity of 146 GB and 16 MB of cache. This is the latest (fully validated) disk model available to DiskSim. Though somewhat old, we decided to use these disks in order to take advantage of the detailed simulation model offered by DiskSim, rather than using a less detailed one. Besides, since our analysis is a comparative one, the disks's performance should benefit or harm all strategies equally. Since the capacity and number of disks in the original traced systems is different from our configuration, we determine the datasets for each trace via static analysis. These datasets are mapped onto the simulated disks uniformly so that all disks have the same access probability.

All the arrays simulated use 50 disks, except those for CRAID-5_d and CRAID-5_d^+ that include 10 additional disks (20%) for the dedicated cache. RAID-5 uses a parity group size of 10 disks both as a stand-alone allocation policy or as a part of a CRAID configuration. Similarly, RAID-5^+ begins with 10 disks and adds a new array of 3, 4, 5, 7, 9 and 12 disks in each expansion step until the 50 disk mark is reached. The stripe unit for all policies is 128 KB and has been computed based on Chen's and Lee's work [29].

We simulate RAID-5 and RAID-5^+ in their ideal state, i.e., when the dataset has been completely restriped. Since CRAID is permanently in an "expansion" phase and sacrifices a small amount of capacity from each disk, for the strategy to be useful its performance should be close to an optimum RAID-5 array, rather than to

Table 5.1: Hit ratio (%) for each cache partition management algorithm

Trace	LRU	LFUDA	GDSF	ARC	WLRU (ω=0.5)
cello99	**65.23**	65.23	48.75	**65.66**	65.22
deasna	89.63	**89.90**	67.24	89.65	**89.73**
home02	**93.91**	93.86	77.93	**93.92**	93.90
webresearch	81.14	78.92	54.41	**82.38**	**82.14**
webusers	80.40	78.72	60.49	**81.01**	**81.40**
wdev	91.04	**91.88**	32.78	**91.06**	91.02
proj	75.55	75.73	25.43	**75.58**	**75.65**

one being restriped. In the remainder of this section we present the summarized results of our experimental runs. In all experiments, the cache partition begins in a cold state.

5.4 Management of the Cache Partition

Here we evaluate the effectiveness of the different cache management algorithms supported by the I/O Monitor (refer to Section 5.2.2112). In this experiment we are concerned with the ideal results of the prediction algorithms to select the best one for CRAID. Thus, we use a simplified disk model that resolves each I/O instantly, and allows us to measure the properties of each algorithm with no interferences. The remaining experiments use the more realistic disk model.

Tables 5.1↑ and 5.2122 show, respectively, the hit and replacement ratio delivered by each algorithm using a P_C size of 0.1% the weekly working set. We observe that all algorithms but one behave similarly, with the ARC algorithm showing the best results in both evaluations. The only exception is the GDSF algorithm, which shows significantly worse results probably due to the addition of the request size as a metric, which is not very useful in this kind of scenario.

For CRAID-5* strategies, however, evictions of clean blocks are preferred as long as the effectiveness of the algorithm is not compromised. This is because evicting a dirty block forces CRAID to update the original blocks and its parity in the P_A, which requires 4 additional I/Os (2 reads and 2 writes).

For this reason, in the CRAID-5* experiments we configure the I/O Monitor with the WLRU ($\omega = 0.5$) algorithm since it shows a hit and replacement ratio similar to ARC, while reducing the amount of dirty evictions. For the CRAID-0 experiments,

Table 5.2: Replacement ratio (%) for each cache partition management algorithm

Trace	LRU	LFUDA	GDSF	ARC	WLRU (ω=0.5)
cello99	34.76	34.76	51.24	**34.31**	**33.76**
deasna	10.36	**10.09**	32.74	10.34	**10.34**
home02	6.08	6.13	22.06	**6.07**	**6.08**
webresearch	18.84	21.06	45.58	**17.60**	**18.83**
webusers	19.58	21.26	39.50	**18.98**	**19.28**
wdev	8.88	**8.04**	67.13	8.85	**8.58**
proj	24.42	**24.24**	74.55	**24.39**	24.72

however, we favor a pure LRU algorithm because its implementation is simpler than that of ARC, which is more desirable in a RAID controller.

5.5 CRAID-5* Response Time

In this section we evaluate the performance impact of using CRAID with parities: for each allocation policy and configuration, we measure the response time of each read and write request occurred during the simulations. Figures 5.6123 and 5.8126 show the response time measurements[6] of each CRAID variant, compared to the RAID-5 and RAID-5+ layouts.

[6] *Computed with a 95% confidence interval.*

Note that each strategy was simulated with different cache partition sizes in order to estimate the influence of this parameter on performance. In the results shown in this section, the cache partition is successively doubled until no evictions have to be performed. This represents the best case for CRAID since data movement between the partitions is reduced to a minimum.

5.5.1 Read requests

The results for read requests are shown in Figure 5.6↷. First, we observe that requests take notably longer to complete in RAID-5+ than in RAID-5 in all cases. This is to be expected since the longer stripes in RAID-5 increase its potential parallelism and provide a more effective workload distribution.

Second, in most traces, hybrid strategies CRAID-5 and CRAID-5+ offer performance comparable to that of vanilla RAID-5, and even better for certain cache sizes (e.g., the webusers trace in Figure 5.6d↷). The explanation lies in the fact that

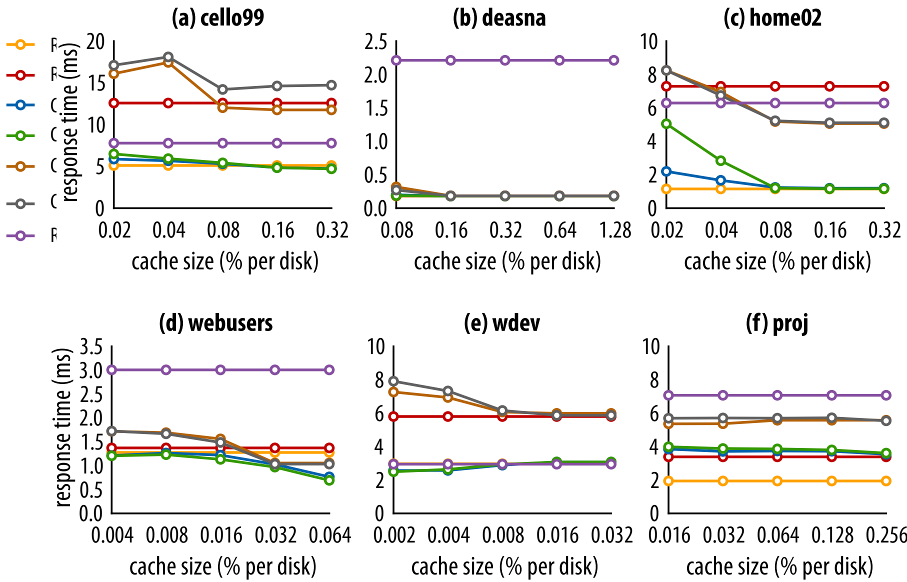


Figure 5.6: I/O response time for read requests in CRAID-5* strategies. The experiment simulates all individual user requests collected in the traces and measures the time taken for each read request to complete. The points in the figure depict the mean response time for all user requests computed with a 95% confidence interval. Note that errorbars are not included since the error interval is too small to be meaningful.

CRAID's cache partition is able to better exploit the spatial locality available in commonly used data: co-locating hot data in a small area of each disk helps reduce seek times when compared to the same data being randomly spread over the entire disk. This is proven by the results shown in Figure 5.7$_{124}$: this figure shows the probability distribution (CDF) of the *sequential access percentage* for the cello99 and webusers traces (computed as *#seq_access/#accesses* and aggregated per second of simulation). Here we see that access sequentiality in CRAID-5 and CRAID-5^{+} is similar to that of RAID-5 and significantly better than that of the RAID-5^{+} strategy.

Nevertheless, the effectiveness of the strategy depends on how well hot data is predicted. Figure 5.6f$_{\uparrow}$ shows that performance results for the proj trace are not as good as in the other traces. Table 5.3$_{124}$ shows that CRAID's best hit ratio for the proj trace is lower than in other traces (85.25% vs. 99.51% in home02, for instance) and that its eviction count is higher. These two factors contribute to more data being transferred to the cache partition and explain the drop in performance.

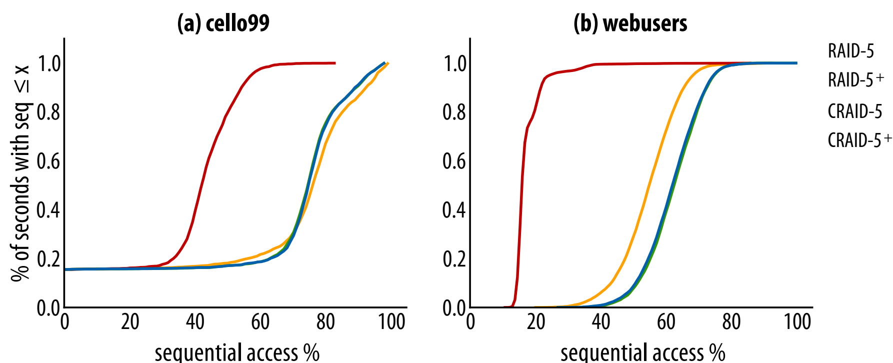


Figure 5.7: Cumulative distribution functions of sequential accesses in traces cello99 and webusers. Sequentiality percentages are computed as $\frac{\#seq_access}{\#accesses}$ and captured at each second of simulation. Simulations themselves are configured with P_C size of 0.02% and 0.004%, respectively. Similar results were observed for the other traces.

Table 5.3: Best hit ratio and worst eviction ratio from all simulations

Trace	Best hit ratio		Worst eviction ratio	
	Reads	Writes	Reads	Writes
cello99	97.85%	98.88%	21.28%	9.53%
deasna	99.53%	97.80%	0.92%	3.17%
home02	99.51%	99.53%	3.32%	2.59%
webresearch	-	98.76%	-	7.66%
webusers	94.95%	99.33%	16.65%	6.56%
wdev	98.62%	99.40%	1.90%	10.76%
proj	85.25%	88.45%	21.97%	9.13%

Also notice that, most interestingly, the performance and sequentiality provided by CRAID-5$^+$ is similar to that of CRAID-5, even though it uses a RAID-5$^+$ strategy for the archive partition. This proves that the cache partition is absorbing most of the I/O, and the array behaves like an ideal RAID-5 array independently of the strategy used for stale data.

Third, increasing the size of the cache partition improves read response times in all CRAID-5 variants. This is to be expected since a larger cache partition increases the probability of a cache hit and also decreases the number of evictions, which greatly improves the effectiveness of the strategy. In most traces, however, once a certain partition size S_{max} is reached, response times stop improving (e.g., deasna

Table 5.4: Comparison of CRAID's dedicated vs. non-dedicated approach for the wdev trace (P_C size: 0.002%). Similar results were observed for the other traces.

Strategy	Mean		99th pctile		Max	
	Ioq	Conc	Ioq	Conc	Ioq	Conc
CRAID-5^+_d	4.74	6.49	63	23	807	40
CRAID-5^+	2.11	8.65	20	44	381	50

Ioq: ioqueue size, Conc: concurrent disks.

with S_{max} = 0.16% or homeo2 with S_{max} = 0.08%, Figures 5.6b and 5.6c123, respectively). Examination of these traces shows that CRAID is able to deliver a near maximum hit ratio with a partition of size S_{max}, and increasing it further provides barely noticeable benefits.

Finally, we see that using dedicated disks for the cache partition significantly degrades the performance offered by CRAID. Examination of the traces reveals that, as expected, the I/O queues in the dedicated disks have significantly more pending requests than those in the disks of the non-dedicated strategies. Also, the number of concurrently active disks during the simulation is lower (see Table 5.4↑). These results confirm our expectations that using a part of each disk to create the cache partition is better than using dedicated disks (elaborated in Section 5.2.1110).

Interestingly, the results from Random Slicing with parity support ($RandSl_p$) are usually faster than RAID-5^+ but slower than RAID-5[7] and are always worse than the results of the non-dedicated CRAID strategies. This is to be expected since a randomized distribution can disrupt locality patterns and the maximum parallelism obtained by this kind of distribution will be lower than that of an adequately distributed RAID in most cases.

[7] *Except in those workloads where the two RAID variants show similar results.*

5.5.2 Write requests

The results for write requests are shown in Figure 5.8126. Similarly to read requests, we observe that write requests are significantly slower in RAID-5^+ than in RAID-5, for all traces. Most importantly, the hybrid strategies CRAID-5 and CRAID-5^+ perform better than traditional RAID-5 in all traces except webusers, where performance is slightly below that of RAID-5.

These improved response times can be explained by two reasons. First, since write requests are always served from the cache partition (except in the case of

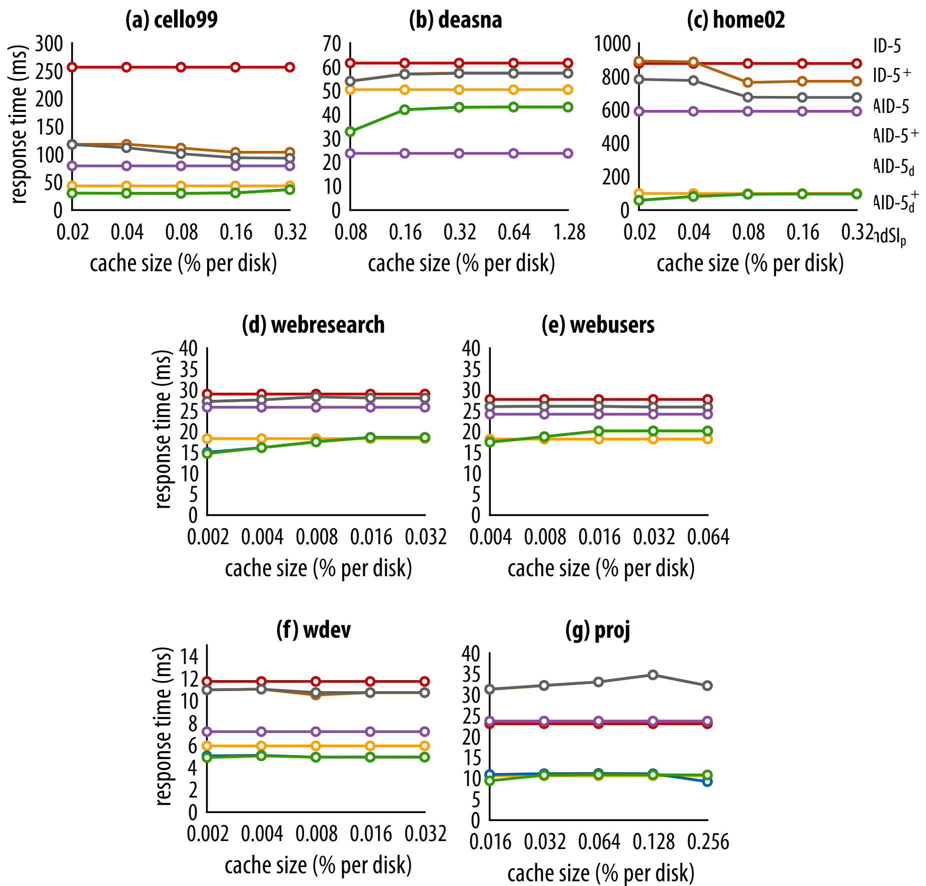


Figure 5.8: I/O response time for write requests in CRAID-5* strategies. The experiment simulates all individual user requests collected in the traces and measures the time taken for each write request to complete. The points in the figure depict the mean response time for all user requests computed with a 95% confidence interval. Note that errorbars are not included since the error interval is too small to be meaningful.

an eviction), response times benefit greatly from the improved spatial locality and sequentiality provided by the cache partition.[8] Second, the smaller the cache partition fragment for each disk is, the more likely it is that accesses to this fragment benefit from the disk's internal cache. This explains why response times in Figure 5.8↑ increase slightly for larger partition sizes: a smaller cache partition means more evictions in CRAID, but it also means a smaller fragment for each disk and a

[8] *Obviously, provided that the prediction of the working set is accurate.*

more effective use of its internal cache. The effect of this internal cache is highly beneficial, to the point that it amortizes the additional work produced by extra evictions.

Dedicated strategies CRAID-5$_d$ and CRAID-5$_d^+$, on the other hand, show worse response times when compared to their non-dedicated counterparts, exactly by the same causes explained for read requests.

Concerning Random Slicing, the results show the same pattern observed for read requests: it is usually faster than RAID-5$^+$ but slower than RAID-5, and the non-dedicated variants of CRAID behave better in most cases.

5.6 CRAID-5* Workload Distribution

In this experiment we evaluate the ability of CRAID-5* strategies to maintain a uniform workload distribution. For each second of simulation we measure the I/O load in MBs received by each disk and we compute the *coefficient of variation* as a metric to evaluate the uniformity of its distribution. The coefficient of variation (c_v) expresses the standard deviation as a percentage of the average (σ/μ), and can be interpreted as how the actual workload deviates from an ideal distribution.[9] We perform this experiment for all strategies already described and with all the partition sizes used in Section 5.5$_{122}$.

[9] *The smaller c_v is, the more uniform the data distribution.*

5.6.1 Impact of CRAID

Figures 5.9a and 5.9b$_{128}$ show CDFs of c_v per % of samples (seconds) for the deasna and wdev traces, respectively. Notice that for CRAID strategies we show both the *best* and *worst* curves obtained (labeled as 'b.c' and 'w.c', respectively) and we compare them with the results for RAID-5 and RAID-5$^+$ (see Table 5.5$_{129}$ for their respective correspondences with actual partition sizes). We observe that there is a significant difference between the workload distribution offered by RAID-5 and that of the RAID-5$^+$ layout, which is to be expected since the "segmented" nature of RAID-5$^+$ naturally hinders a uniform workload distribution.

Most interestingly, all CRAID strategies exhibit a workload distribution very similar to (and sometimes better than) that of RAID-5. More importantly, this benefit appears in even those CRAID configurations that use RAID-5$^+$ for the archive partition, despite its poor performance and uneven distribution. This definitely proves

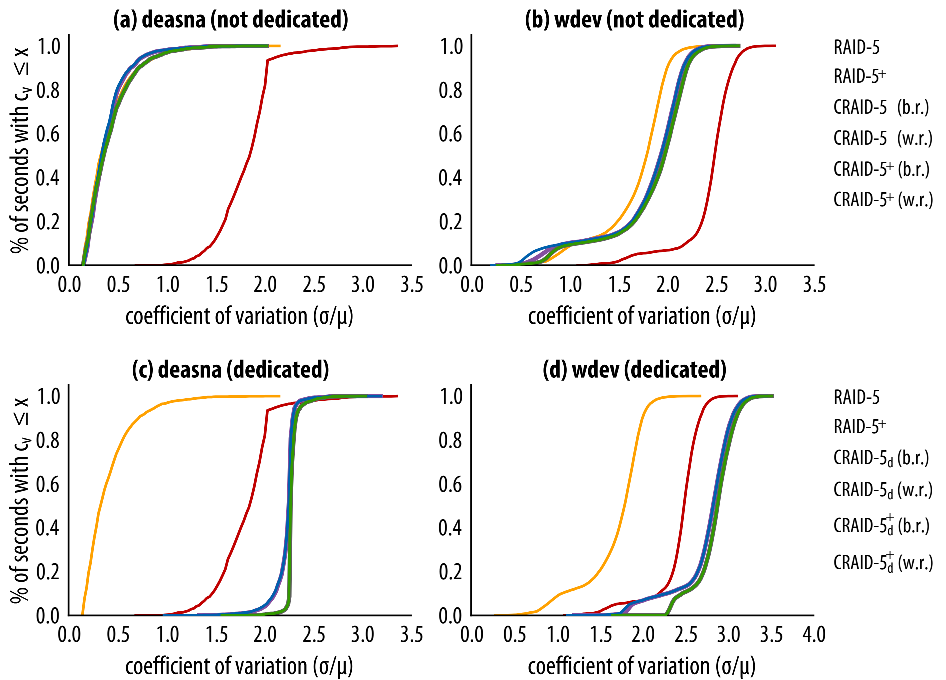


Figure 5.9: Cumulative distribution functions of the coefficient of variation (c_v) for workload distribution in traces deasna and wdev. Results were collected during simulations (1-week) and aggregated per % of samples (seconds). Note that curves that follow the same path as others are drawn with a thicker line width in order to be distinguishable. Similar results were observed for the other traces.

that the cache partition is successful in absorbing most I/O, and that it behaves close to an ideal RAID-5 despite the cost of the additional data transfers.

5.6.2 Influence of the cache partition size

Though barely noticeable, an unexpected result is that, in all traces, the workload distribution degrades as the cache partition grows (see Table 5.5↷). Examination of the traces shows that a larger cache partition slightly increases the probability that certain subsets of disks are more used than others due to the different layout of data blocks. This is reasonable since the current version of CRAID does not perform direct actions to enforce a certain workload distribution, but rather relies on the

Table 5.5: Influence of P_C size on workload distribution

Trace	CRAID-5		CRAID-5$^+$	
	best c_v	worst c_v	best c_v	worst c_v
cello99	0.02%	0.32%	0.02%	0.32%
deasna	0.08%	1.28%	0.08%	1.28%
home02	0.02%	0.32%	0.02%	0.32%
webresearch	0.002%	0.032%	0.002%	0.032%
webusers	0.004%	0.064%	0.004%	0.064%
wdev	0.002%	0.032%	0.002%	0.032%
proj	0.016%	0.256%	0.016%	0.256%

strategy used for the cache partition. Improving CRAID to employ workload-aware layouts is one of the subjects of our future investigation.

5.6.3 Workload with dedicated disks

The curves for CRAID-5_d and CRAID-5_d^+ seen in Figures 5.9c and 5.9d↶ demonstrate a poor workload distribution for dedicated approaches, when compared to the other strategies. This is to be expected since the dedicated disks absorb much of the I/O workload and end up degrading the global workload of the system. Note that this does not necessarily mean that the workload directed to the dedicated disks is unbalanced, but rather that the other disks are being underutilized.

5.7 Performance of CRAID-0*

Finally, to conclude CRAID's experimental evaluation, let's take a look to the results of CRAID-0. For brevity's sake, we only review the simulations of some of the traces, since the results and conclusions are similar to those shown in the CRAID-5* evaluation. Please note that all the experiments discussed in this section use the P_C size that produced the best results for the CRAID-5* experiments.

5.7.1 Response time

The bar plots in Figure 5.10130 (top row) show the average response time[10] for read requests for the cello99, deasna, home02 and webusers traces. As expected, requests

[10] *Again, computed with a 95% confidence interval.*

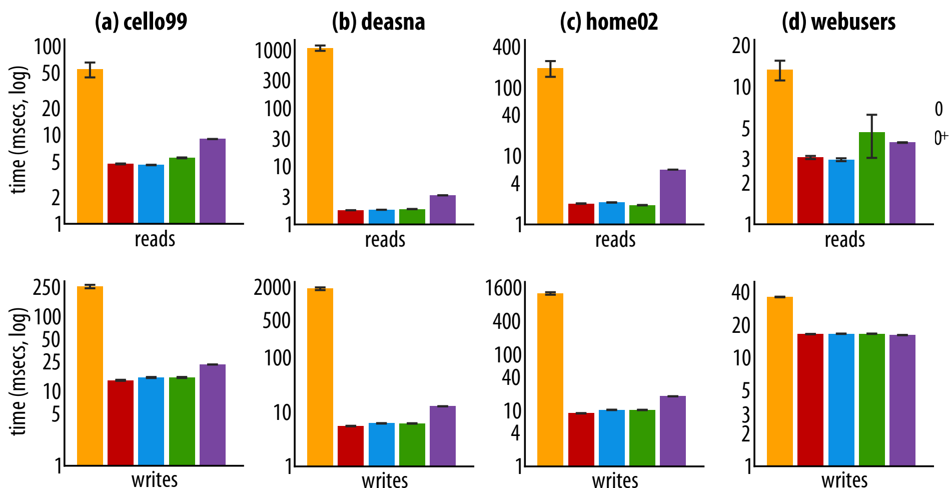


Figure 5.10: Average I/O response time in CRAID-0* strategies. The experiment simulates all individual user requests collected in the traces and measures the time taken for each write request to complete. Bars depict the mean response time for all user requests computed with a 95% confidence interval.

are significantly slower in SEQ than in RAID-0, with response times in general between one and two orders of magnitude slower. Notice that in most cases, the performance of CRAID-0^+ is similar to that of RAID-0, which further validates our hypothesis that there's no need to optimize the data placement of all the data space, since CRAID is able to obtain good results even with a bad archival strategy.

In addition, notice that the performance of CRAID-0 shown for traces cello99 and webusers is slightly better than that of RAID-0. Again, this happens because currently active data is clustered in the cache partition, thus benefiting from improved spatial locality. Interestingly, the performance gain for webusers is lower than for others workloads.

Plots in the bottom row show the average response time for writes requests. We can see the same behavior than in the previous experiment: the response times of the hybrid strategies are similar to those of RAID-0 even taking into account the simple LRU replacement policy and the extra overhead added by replacing data from the cache partition.

The results for Random Slicing are similar to those we saw in Section 5.5_{122}. As expected, the performance exhibited by Random Slicing is much better than that of

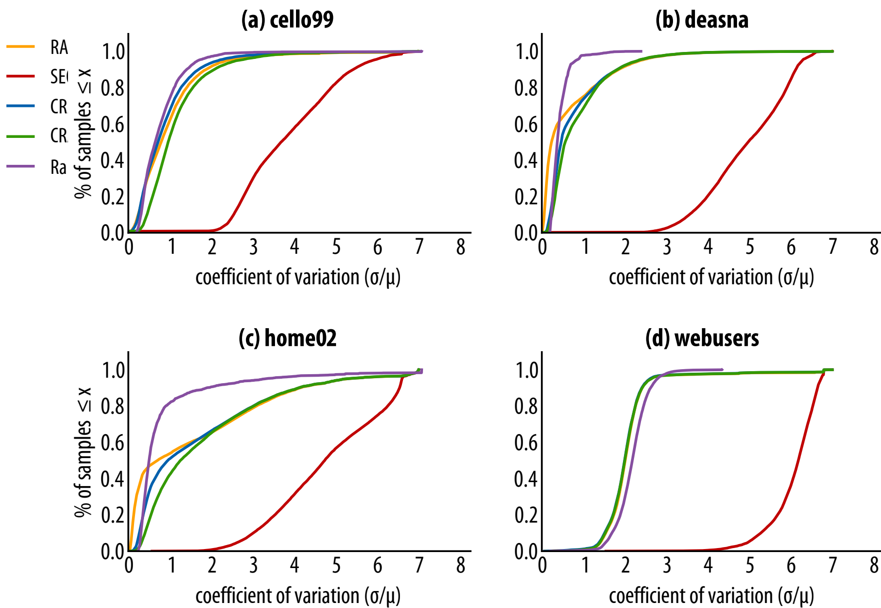


Figure 5.11: Cumulative distribution function of the coefficient of variation (c_v) for workload distribution in traces cello99, deasna, home02 and webusers. Results were collected during simulations (1-week) and aggregated per % of samples (seconds).

SEQ, but in general requests are slower than RAID-0 and the CRAID-0 variants. The reasoning behind this behavior is that since Random Slicing does not take any steps to preserve locality patterns, it is highly possible that the randomization process degrades spatial locality.

5.7.2 Workload distribution

CDFs in Figure 5.11↑ show the probability distribution of the coefficient of variation for each simulation. As expected, SEQ offers highly unbalanced I/O in all simulations, whereas the workload distribution offered by RAID-0 is significantly better in all cases, even when using SEQ as the archival strategy.

Interestingly, the workload distribution shown by Random Slicing is slightly better than the exhibited by RAID-based strategies. Clearly, the workload distribution provided by the use of a pseudo-random hash function is closer to an ideal distri-

bution than RAID striping. The reason for this lies in that access patterns to certain clusters of blocks can be better spread by a randomization function, whereas a normal striping can be liable to use only a certain number of devices. Nevertheless, the difference is only significant in the home02 trace (Figure 5.11c$_{131}$), and the results offered by RAID-0 and the CRAID-0* variants are still good considering the performance improvement.

5.8 Conclusions

In this chapter, we have proposed and evaluated CRAID, a self-optimizing RAID architecture that automatically rebalances the layout of frequently accessed data in a dedicated cache partition. CRAID is designed to accelerate the upgrade process of traditional RAID architectures by limiting it to this partition, which contains the data that is currently important and on which certain QoS levels must be kept.

We have analyzed CRAID using seven real-world traces each with differing workloads and collected at different times during the last decade. Our analysis shows that CRAID is highly successful in predicting the data workload and its variations both in RAID-5 and RAID-0 arrays. Further, if an appropriate data distribution is used for the cache partition, CRAID optimizes the performance of read and write traffic due to the increased locality and sequentiality of frequently accessed data. Specifically, with our experiments we have shown that it is possible to achieve a QoS competitive with an ideal RAID-5 or RAID-0 array by creating a small partition of 1.28% the available storage size or less, regardless of the layout used for cold data within the archive partition. This is particularly important, because it proves that it is possible to archive data incrementally with no special treatment and, at the same time, create specific optimizations for heavily accessed data so that the throughput of the storage architecture scales gracefully.

Additionally, we have also compared CRAID against our previous proposal Random Slicing. Despite redistributing in all cases less data than the ideal amount, CRAID consistently improves the response time of Random Slicing, regardless of the trace examined, both in its RAID-0 and RAID-5 variants. Concerning workload distribution, Random Slicing offers results closer to an ideal distribution than those of CRAID, due to its randomized approach. Nevertheless, the difference is minimal, and the distribution offered by CRAID is competitive with RAID.

In summary, we believe that CRAID is a novel approach to building RAID architectures that can offer reduced upgrade times and I/O performance improvements. In

addition, its ability to combine different data layouts can serve as a starting point to design scalable allocation strategies more conscious about the semantics and correlations between different data blocks.

Chapter 6

Conclusion

"Let the future tell the truth, and evaluate each one according to his work and accomplishments. The present is theirs; the future, for which I have really worked, is mine."
— Nikola Tesla

Dear reader, if you have painstakingly worked your way through all or most of the pages in this dissertation let me thank you and salute you: it is no meager task. In this final chapter of the thesis, we will muse upon the topics discussed in the main chapters and highlight the principal contributions of this work. Finally, we will try to answer one of the most dreaded questions for any scientist: *"what's next?"*

In Chapter 0, '*Humans and information storage*'$_{1}$, we presented an overview of the problem that continuous data creation poses for current and future storage systems. This chapter served as a starting point to discuss the reasons that lead to the increasing creation and storage of data, the intrinsic challenges tied to the management of large amounts of data and the technical limitations of magnetic disks which are still the dominant technology used in large-scale storage.

With Chapter 1, '*Background and related work*'$_{11}$, we gave an overview of the current state-of-the-art knowledge in large-scale storage infrastructures. The primary intention of this chapter was to establish a necessary knowledge basis for the

rest of the thesis, and also to get deeper into the specific problems in large-scale data management, abandoning the necessarily shallow views presented in the first chapter. It was also important to introduce the technological foundations on which later chapters are based, and thus we reviewed the most common storage architectures (like SAN and NAS), as well as the data distribution mechanisms (randomized versus deterministic) currently in use. The chapter concluded by discussing the characterization of access patterns and hierarchical storage which were necessary for Chapter 4_{83} and Chapter 5_{107}, respectively.

Chapter 2, '*Scalable data distribution*'$_{29}$ reported on our research in scalable storage infrastructures based on pseudo-randomized data distribution. In this chapter, the emphasis was put on attaining a *balanced* data distribution with *minimal* data migration during storage upgrades. Although randomized data distribution is a commonly used technique to solve this particular problem and differing strategies abound, there were no formal comparative studies between them before our work. Thus, we offered an analysis that compared some of the best known randomized data distribution strategies in a common environment, evaluating them in terms of distribution fairness, performance, memory usage and adaptivity to changes in the infrastructure.

In addition, we also contributed a definition for a new pseudo-randomized strategy that outperformed the current ones, both in memory usage and lookup time, and had comparable adaptivity. What was novel about this strategy is that it used a small interval table to keep track of changes in the storage infrastructure, thus decoupling the lookup process (which was based on a random function) from the reconfiguration process (which was deterministic).

With Chapter 3, '*PRNGs in data distribution*'$_{65}$, we dived into the field of pseudo-random number generation and how it might affect data distribution. Our main contribution in this chapter was an analysis of eighteen different pseudo-random number generators and their influence on the performance and distribution quality of several randomized strategies. We also contributed a ranking of all the analyzed PRNGs based on how they affected load balance and performance.

The previously discussed chapter marked the end of the first part of the thesis, where we established that minimal data migration is necessary in order to achieve a sustainable growth. With the second part of the thesis, we introduced the notion that minimum data migration might be too costly for storage infrastructures of Petascale and above complexity. This apparent conundrum led us to propose a novel concept for data migration that used semantic information in order to determine *favorable data* to migrate. Hence, instead of looking at the dataset as an opaque slab of data that needs to be sliced and redistributed, we inspected its individual components and redistributed only those that could maximize the future overall performance of the system.

In Chapter 4, '*Long-term locality in mass storage*'$_{83}$, we showed our attempts to find a group of *long-term, frequently used data*; that is, a group of data that received the majority of client accesses and remained stable over time. Our hypothesis was that if such a group existed in a large amount of workloads, it would be possible to design an upgrade strategy that focused only on this data, reducing the amount of work to be done during upgrades and with—hopefully—performance levels similar to other storage strategies.

The chapter contributed an analysis of long-term access patterns in eight storage workloads, which was later extended to four additional workloads. The analysis demonstrated that our initial hypothesis was valid: most workloads showed a high amount of blocks shared during relatively long periods of time, and those blocks received the majority of client accesses. Furthermore, this set of "interesting" blocks was small when compared to the entire dataset. These discoveries are what led us to the final chapter, which finally used the knowledge learned to implement a usage-aware distribution strategy.

Finally, with Chapter 5, '*Extensibility in RAID architectures*'$_{107}$, we presented the results of applying data semantics to the upgrade process of two RAID architectures, RAID-0 and RAID-5. Our proposed solution, CRAID, tracked frequently-used blocks and redistributed them—in real-time—into a dedicated caching partition created from regions in all disks. We analyzed CRAID's performance and workload distribution using one of the most detailed device simulators available, which was fed with real-world traces from seven different environments.

Our experiments with CRAID demonstrated that it is possible to achieve performance levels comparable to those of RAID-0 and RAID-5—and also adapt to changes in the storage infrastructure—only with the on-line migration of frequently-used data. Furthermore, the largest cache partition used in our simulations, and thus the amount of data that needs to be migrated, accounted for 1.28% of the storage capacity, which significantly reduces the migration cost and is an acceptable trade-off in a current storage infrastructure. Finally, we proved that CRAID delivered better response times and a similar workload distribution than our previous proposal, Random Slicing, despite redistributing significantly less data.

Concerning future research lines, the work presented in this dissertation can be developed extensively. While the current CRAID prototype has served to verify that it is possible to amortize the cost of a RAID upgrade by using knowledge about hot data blocks, it uses simple algorithms for prediction and expansion, and it might be worth improving them. For instance, the prototype currently invalidates the entire cache partition when new disks are added, a process which scraps all optimizations done up to that point. Though this benefits the parallelism of the data distribution and new disks can be used immediately, the current strategy was devised to test if our hypothesis held in the simplest case, without complex algorithms. Since in most data workloads the working sets should not change dramatically over time, CRAID could benefit greatly from strategies to rebalance the small amount of data held in the cache partition more intelligently, like those presented in Section 1.5.223. Similarly, the current CRAID prototype does not make any effort to allocate related blocks close to each other. Alternate layout strategies more focused on preserving semantic relations between blocks might yield greater benefits. For instance, it might be interesting to evaluate the effect of copying entire stripes to the cache partition as a way to preserve spatial locality. Besides, this could help reduce the number of parity computations, thus reducing the background I/O present in the array.

Additionally, it would be an interesting challenge to extend the idea behind CRAID to pseudo-randomized data layouts, and add data semantics awareness to Random Slicing in order to reduce the amount of rebalancing between devices. Randomized data distributions are more flexible in nature than deterministic solutions like RAID, and we envision that they may benefit significantly from an enhanced perspective on the data they manage.

Furthermore, the study on data semantics can be significantly expanded. In this dissertation we have only considered the usage of data blocks, but we have not taken into account the interrelations that exist between them. Successfully predicting these relations could, for instance, allow migration algorithms to determine the interesting working sets faster and physically allocate semantic groups of data. For instance, the current version of CRAID relies on the fact that blocks accessed consecutively in a short period of time tend to be related. Using techniques to detect block correlations could improve CRAID and scalable layouts significantly, allowing hot data migration before it is actually needed. Regarding this idea, a line of research we are particularly interested in, is to explore the applicability of machine learning techniques to this kind of detection. We believe that unsupervised learning techniques could be used to model this interrelations, and neural networks could be constructed to infer, learn and take advantage of these access patterns.

While our experiences with CRAID have been positive in RAID-0 and RAID-5 storage, we believe that they can also be applied to RAID-6 or more general erasure codes ($k > 2$), since the overall principle still applies: rebalancing hot data should require less work than producing an ideal distribution. The main caveat of our solution, however, is the cost of additional parity computations and I/O operations for dirty blocks, which directly increases with the number of parity blocks required. It would be interesting to explore if, in these data distributions, the extension cost can also be leveraged by the performance benefits obtained with separate hot-data management, as well as up to which point these results could be generalized.

It would also be interesting to apply our semantic-aware research to power-aware storage. For instance, CRAID could be modified to favor minimal power consumption rather than maximum parallelism and optimal distribution. A preliminary approach, which we have not truly developed yet, could be to confine the caching partition to a dynamically changing subset of disks, thus reducing the power consumption of the remaining disks. Newer metrics would have to be devised in order to adequately balance the performance degradation (due to the restricted partition) and the savings in energy consumption (due to the usage of less disks). Semantic information could also be used to classify data blocks according to their usage schedule and place blocks with similar requirements into the same devices. This would allow the storage architecture to shut down these devices when the "usual usage time" has passed or even to turn them on when they are likely to be used. How this clustering would affect performance, however, has yet to be seen.

Appendix A

The ENT Test for Pseudo-random Sequences

This appendix describes a program, ent, which applies various tests to sequences of bytes stored in files and reports the results of those tests. The program is useful for evaluating pseudo-random number generators for encryption and statistical sampling applications, compression algorithms, and other applications where the information density of a file is of interest.

The ent program performs a variety of tests on the stream of bytes provided and produces output as follows:

```
Entropy = 7.980627 bits per character.

Optimum compression would reduce the size
of this 51768 character file by 0 percent.

Chi square distribution for 51768 samples is 1542.26, and
   randomly
would exceed this value less than 0.01 percent of the times.

Arithmetic mean value of data bytes is 125.93 (127.5=random).
Monte Carlo value for Pi is 3.169834647 (error 0.90 percent).
Serial correlation coefficient is 0.004249 (totally
   uncorrelated=0.0).
```

In the following pages we describe each of the tests performed and the meaning behind the produced values.

- **Entropy test:** The information density of the contents of the sequence, expressed as a number of bits per character. The results above, which resulted from processing an image file compressed with JPEG, indicate that the file is extremely dense in information—essentially random. Hence, compression of the file is unlikely to reduce its size. By contrast, the C source code of the program has entropy of about 4.9 bits per character, indicating that optimal compression of the file would reduce its size by 38% [58, pp. 104–108].

- **Chi-square test:** The Chi-square (χ^2) test is the most commonly used test for the randomness of data, and is extremely sensitive to errors in pseudo-random sequence generators. The χ^2 distribution is calculated for the file's stream of bytes and expressed as an absolute number and a percentage which indicates how frequently a truly random sequence would exceed the value calculated. We interpret the percentage as the degree to which the sequence tested is suspected of being non-random. If the percentage is greater than 99% or less than 1%, the sequence is almost certainly not random. If the percentage is between 99% and 95% or between 1% and 5%, the sequence is suspect. Percentages between 90% and 95% and 5% and 10% indicate the sequence is "almost suspect". Note that our JPEG file, while very dense in information, is far from random as revealed by the χ^2 test.

  Applying this test to the output of various pseudo-random sequence generators is interesting. The low-order 8 bits returned by the standard Unix `rand()` function, for example, yield:

  ```
  Chi square distribution for 500000 samples is 0.01, and
     randomly would exceed this value more than 99.99
     percent of the times.
  ```

  While an improved generator [107] reports:

  ```
  Chi square distribution for 500000 samples is 212.53, and
      randomly would exceed this value 97.53 percent of the
      times.
  ```

Thus, the standard Unix generator (or at least the low-order bytes it returns) is unacceptably non-random, while the improved generator is much better but still sufficiently non-random to cause concern for demanding applications. Contrast both of these software generators with the χ^2 result of a genuine random sequence created by timing radioactive decay events:

```
Chi square distribution for 500000 samples is 249.51, and
    randomly would exceed this value 40.98 percent of the
    times.
```

See Knuth's *"The Art of Computer Programming, Volume 2: Seminumerical Algorithms"* pp. 35–40 [72] for more information on the χ^2 test.

- **Arithmetic mean test:** This is simply the result of summing the all the bytes in the sequence and dividing by the sequence length. If the data are close to random, this should be about 127.5 (0.5 if individual bits are considered). If the mean departs from this value, the values are consistently high or low.

- **Monte Carlo Value for π:** Each successive sequence of six bytes is used as 24 bit x and y coordinates within a square. If the distance of the randomly-generated point is less than the radius of a circle inscribed within the square, the six-byte sequence is considered a "hit". The percentage of hits can be used to calculate the value of π. For very large streams (this approximation converges very slowly), the value will approach the correct value of π if the sequence is close to random. A 500,000 byte sequence created by radioactive decay yielded:

```
Monte Carlo value for Pi is 3.14358057 (error 0.06
   percent).
```

- **Serial correlation coefficient test:** This value measures the extent to which each byte in the sequence depends upon the previous byte. For random sequences, this value (which can be positive or negative) will, of course, be close to zero. A non-random byte stream such as a C program will yield a serial correlation coefficient on the order of 0.5. Wildly predictable data such as uncompressed bitmaps will exhibit serial correlation coefficients approaching 1. Again, refer to Knuth's *"The Art of Computer Programming, Volume 2: Seminumerical Algorithms"* pp. 35–40 [72] for more details.

Appendix B

The NIST Test Suite

The NIST Test Suite is a statistical package consisting of 15 tests that were developed to test the randomness of (arbitrarily long) binary sequences produced by either hardware or software based cryptographic random or pseudo-random number generators. These tests focus on a variety of different types of non-randomness that could exist in a sequence. Some tests are decomposable into a variety of subtests.

A number of tests in the test suite have the *standard normal* and the *chi-square* (χ^2) as reference distributions. If the sequence under test is in fact non-random, the calculated test statistic will fall in extreme regions of the reference distribution. The standard normal distribution (i.e., the bell-shaped curve) is used to compare the value of the test statistic obtained from the random number generator with the expected value of the statistic under the assumption of randomness. The test statistic for the standard normal distribution is of the form $z = (x - \mu)/\sigma$, where x is the sample test statistic value, and μ and σ^2 are the expected value and the variance of the test statistic. The χ^2 distribution (i.e. a left skewed curve) is used to compare the goodness-of-fit of the observed frequencies of a sample measure to the corresponding expected frequencies of the hypothesized distribution. The test statistic is of the form:

$$\chi^2 = \sum \left(\frac{(o_i - e_i)^2}{e_i} \right), \tag{B.1}$$

where o_i and e_i are the observed and expected frequencies of occurrence of the measure, respectively.

For many of the tests in this test suite, the assumption has been made that the size of the sequence length, n, is large (of the order 10^3 to 10^7). For such large sample sizes of n, asymptotic reference distributions have been derived and applied to carry out the tests. Most of the tests are applicable for smaller values of n. However, if used for smaller values of n, the asymptotic reference distributions would be inappropriate and would need to be replaced by exact distributions that would commonly be difficult to compute.

B.1 Frequency (Monobit) Test

The focus of the test is the proportion of zeroes and ones for the entire sequence. The purpose of this test is to determine whether the number of ones and zeros in a sequence are approximately the same as would be expected for a truly random sequence. The test assesses the closeness of the fraction of ones to 1/2, that is, the number of ones and zeroes in a sequence should be about the same. All subsequent tests depend on the passing of this test.

Reference distribution	half-normal distribution
Decision Rule (at the 1% Level)	$p < 0.01 \Rightarrow X_n$ non-random

References for Test

[1] Kai Lai Chung and Farid AitSahlia. *Elementary Probability Theory: With Stochastic Processes and an Introduction to Mathematical Finance*. Springer, 2003

[2] Jim Pitman. *Probability*. Springer New York, 1993, 93–108

B.2 Frequency Test within a Block

The focus of the test is the proportion of ones within m-bit blocks. The purpose of this test is to determine whether the frequency of ones in an m-bit block is approx-

imately m/2, as would be expected under an assumption of randomness. For block size $m = 1$, this test degenerates to test 1, the Frequency (Monobit) Test.

Reference distribution	χ^2 distribution
Decision Rule (at the 1% Level)	$p < 0.01 \Rightarrow X_n$ non-random

References for Test

[1] Milton Abramowitz and Irene A. Stegun. *Handbook of Mathematical Functions: With Formulars, Graphs, and Mathematical Tables*. Vol. 55. DoverPublications.com, 1964

[2] D.E. Knuth. "Volume 2: Seminumerical Algorithms". *The Art of Computer Programming* (1997), 192

[3] Nick Maclaren. "Cryptographic Pseudo-random Numbers in Simulation". *Fast Software Encryption*. Springer, 185–190

B.3 Runs Test

The focus of this test is the total number of runs in the sequence, where a run is an uninterrupted sequence of identical bits. A run of length k consists of exactly k identical bits and is bounded before and after with a bit of the opposite value. The purpose of the runs test is to determine whether the number of runs of ones and zeros of various lengths is as expected for a random sequence. In particular, this test determines whether the oscillation between such zeros and ones is too fast or too slow.

Reference distribution	χ^2 distribution
Decision Rule (at the 1% Level)	$p < 0.01 \Rightarrow X_n$ non-random

References for Test

[1] Jean Dickinson Gibbons and Subhabrata Chakraborti. *Nonparametric statistical inference*. Vol. 168. CRC press, 2003

[2] Anant P. Godbole and Stavros G. Papastavridis. *Runs and patterns in probability: Selected papers*. Vol. 283. Springer, 1994

B.4 Test for the Longest Run of Ones in a Block

The focus of the test is the longest run of ones within m-bit blocks. The purpose of this test is to determine whether the length of the longest run of ones within the tested sequence is consistent with the length of the longest run of ones that would be expected in a random sequence. Note that an irregularity in the expected length of the longest run of ones implies that there is also an irregularity in the expected length of the longest run of zeroes. Therefore, only a test for ones is necessary.

Reference distribution	χ^2 distribution
Decision Rule (at the 1% Level)	$p < 0.01 \Rightarrow X_n$ non-random

References for Test

[1] Florence Nightingale David and David Elliot Barton. *Combinatorial chance*. Griffin London, 1962

[2] Pál Révész. *Random walk in random and non-random environments*. World Scientific, 2005

B.5 Binary Matrix Rank Test

The focus of the test is the rank of disjoint sub-matrices of the entire sequence. The purpose of this test is to check for linear dependence among fixed length substrings of the original sequence.

Reference distribution	χ^2 distribution
Decision Rule (at the 1% Level)	$p < 0.01 \Rightarrow X_n$ non-random

References for Test

[1] I.N. Kovalenko. "Distribution of the linear rank of a random matrix". *Theory of Probability & Its Applications* 17.2 (1973), 342–346

[2] George Marsaglia. "DIEHARD: a battery of tests of randomness". *Online: http://www.stat.fsu.edu/pub/diehard/* (1996)

[3] George Marsaglia and Liang-Huei Tsay. "Matrices and the structure of random number sequences". *Linear algebra and its applications* 67 (1985), 147–156

B.6 Discrete Fourier Transform (Spectral) Test

The focus of this test is the peak heights in the Discrete Fourier Transform of the sequence. The purpose of this test is to detect periodic features (i.e., repetitive patterns that are near each other) in the tested sequence that would indicate a deviation from the assumption of randomness. The intention is to detect whether the number of peaks exceeding the 95% threshold is significantly different than 5%.

Reference distribution normal distribution

Decision Rule (at the 1% Level) $p < 0.01 \Rightarrow X_n$ non-random

References for Test

[1] Ronald Newbold Bracewell. *The Fourier transform and its applications*. Vol. 31999. McGraw-Hill New York, 1986

[2] Song-Ju Kim, Ken Umeno, and Akio Hasegawa. "Corrections of the NIST statistical test suite for randomness". *arXiv preprint nlin/0401040* (2004)

B.7 Non-overlapping Template Matching Test

The focus of this test is the number of occurrences of pre-specified target strings. The purpose of this test is to detect generators that produce too many occurrences

of a given non-periodic (aperiodic) pattern. For this test and for the Overlapping Template Matching Test of Appendix B.8, an m-bit window is used to search for a specific m-bit pattern. If the pattern is not found, the window slides one bit position. If the pattern is found, the window is reset to the bit after the found pattern, and the search resumes.

Reference distribution χ^2 distribution

Decision Rule (at the 1% Level) $p < 0.01 \Rightarrow X_n$ non-random

References for Test

[1] Andrew D. Barbour, Lars Holst, and Svante Janson. *Poisson approximation*. Clarendon press Oxford, 1992

B.8 Overlapping Template Matching Test

The focus of the Overlapping Template Matching test is the number of occurrences of pre-specified target strings. Both this test and the Non-overlapping Template Matching Test of Appendix B.7 use an m-bit window to search for a specific m-bit pattern. As with the previous test, if the pattern is not found, the window slides one bit position. The difference between this test and the test in Appendix B.7 is that when the pattern is found, the window slides only one bit before resuming the search.

Reference distribution χ^2 distribution

Decision Rule (at the 1% Level) $p < 0.01 \Rightarrow X_n$ non-random

References for Test

[1] O. Chrysaphinou and S. Papastavridis. "A limit theorem on the number of overlapping appearances of a pattern in a sequence of independent trials". *Probability theory and related fields* 79.1 (1988), 129–143

[2] Kenji Hamano and Toshinobu Kaneko. “Correction of overlapping template matching test included in NIST randomness test suite”. *IEICE transactions on fundamentals of electronics, communications and computer sciences* 90.9 (2007), 1788–1792

[3] Norman L. Johnson, Adrienne W. Kemp, and Samuel Kotz. *Univariate discrete distributions*. Vol. 444. John Wiley & Sons, 2005

B.9 Maurer's "Universal Statistical" Test

The focus of this test is the number of bits between matching patterns (a measure that is related to the length of a compressed sequence). The purpose of the test is to detect whether or not the sequence can be significantly compressed without loss of information. A significantly compressible sequence is considered to be non-random.

Reference distribution	half-normal distribution
Decision Rule (at the 1% Level)	$p < 0.01 \Rightarrow X_n$ non-random

References for Test

[1] Jean-Sébastien Coron and David Naccache. “An accurate evaluation of Maurer's universal test”. *Selected Areas in Cryptography*. Springer, 57–71

[2] Helen Gustafson, Ed Dawson, Lauren Nielsen, and W. Caelli. “A computer package for measuring the strength of encryption algorithms”. *Computers & Security* 13.8 (1994), 687–697

[3] Ueli M. Maurer. “A universal statistical test for random bit generators”. *Journal of cryptology* 5.2 (1992), 89–105

[4] Alfred J. Menezes, Paul C. Van Oorschot, and Scott A. Vanstone. *Handbook of applied cryptography*. CRC press, 2010

[5] J. Ziv. “Compression, tests for randomness and estimating the statistical model of an individual sequence”. *Sequences*. Springer, 366–373

[6] Jacob Ziv and Abraham Lempel. “A universal algorithm for sequential data compression”. *Information Theory, IEEE Transactions on* 23.3 (1977), 337–343

B.10 Linear Complexity Test

The focus of this test is the length of a linear feedback shift register (LFSR). The purpose of this test is to determine whether or not the sequence is complex enough to be considered random. Random sequences are characterized by longer LFSRs. An LFSR that is too short implies non-randomness.

Reference distribution χ^2 distribution

Decision Rule (at the 1% Level) $p < 0.01 \Rightarrow X_n$ non-random

References for Test

[1] Helen Gustafson, Ed Dawson, Lauren Nielsen, and W. Caelli. "A computer package for measuring the strength of encryption algorithms". *Computers & Security* 13.8 (1994), 687–697

[2] Alfred J. Menezes, Paul C. Van Oorschot, and Scott A. Vanstone. *Handbook of applied cryptography*. CRC press, 2010

[3] Rainer A. Rueppel. *Analysis and design of stream ciphers*. Springer-Verlag New York, Inc., 1986

B.11 Serial Test

The focus of this test is the frequency of all possible overlapping m-bit patterns across the entire sequence. The purpose of this test is to determine whether the number of occurrences of the 2^m m-bit overlapping patterns is approximately the same as would be expected for a random sequence. Random sequences have uniformity; that is, every m-bit pattern has the same chance of appearing as every other m-bit pattern. Note that for $m = 1$, the Serial test is equivalent to the Frequency Test of Appendix B.1.

Reference distribution χ^2 distribution

Decision Rule (at the 1% Level) $p < 0.01 \Rightarrow X_n$ non-random

References for Test

[1] I.J. Good. "The serial test for sampling numbers and other tests for randomness". *Proceedings of the Cambridge Philosophical Society*. Vol. 49. Cambridge Univ Press, 276–284

[2] M. Kimberley. "Comparison of two statistical tests for keystream sequences". *Electronics Letters* 23.8 (1987), 365–366

[3] D.E. Knuth. "Volume 2: Seminumerical Algorithms". *The Art of Computer Programming* (1997), 192

[4] Alfred J. Menezes, Paul C. Van Oorschot, and Scott A. Vanstone. *Handbook of applied cryptography*. CRC press, 2010

B.12 Approximate Entropy Test

As with the Serial test of Appendix B.11, the focus of this test is the frequency of all possible overlapping m-bit patterns across the entire sequence. The purpose of the test is to compare the frequency of overlapping blocks of two consecutive/adjacent lengths (m and $m + 1$) against the expected result for a random sequence.

Reference distribution	χ^2 distribution
Decision Rule (at the 1% Level)	$p < 0.01 \Rightarrow X_n$ non-random

References for Test

[1] Steve Pincus and Rudolf E. Kalman. "Not all (possibly)"random" sequences are created equal". *Proceedings of the National Academy of Sciences* 94.8 (1997), 3513–3518

[2] Steve Pincus and Burton H. Singer. "Randomness and degrees of irregularity". *Proceedings of the National Academy of Sciences* 93.5 (1996), 2083–2088

[3] Andrew L. Rukhin. "Approximate entropy for testing randomness". *Journal of Applied Probability* 37.1 (2000), 88–100

B.13 Cumulative Sums (Cusum) Test

The focus of this test is the maximal excursion (from zero) of the random walk defined by the cumulative sum of adjusted $(-1, +1)$ digits in the sequence. The purpose of the test is to determine whether the cumulative sum of the partial sequences occurring in the tested sequence is too large or too small relative to the expected behavior of that cumulative sum for random sequences. This cumulative sum may be considered as a random walk. For a random sequence, the excursions of the random walk should be near zero. For certain types of non-random sequences, the excursions of this random walk from zero will be large.

Reference distribution	normal distribution
Decision Rule (at the 1% Level)	$p < 0.01 \Rightarrow X_n$ non-random

References for Test

[1] Pál Révész. *Random walk in random and non-random environments*. World Scientific, 2005

[2] Frank Spitzer. *Principles of random walk*. Vol. 34. Springer, 2001

B.14 Random Excursions Test

The focus of this test is the number of cycles having exactly K visits in a cumulative sum random walk. The cumulative sum random walk is derived from partial sums after the $(0, 1)$ sequence is transferred to the appropriate $(-1, +1)$ sequence. A cycle of a random walk consists of a sequence of steps of unit length taken at random that begin at and return to the origin. The purpose of this test is to determine if the number of visits to a particular state within a cycle deviates from what one would expect for a random sequence. This test is actually a series of eight tests (and conclusions), one test and conclusion for each of the states: -4, -3, -2, -1 and +1, +2, +3, +4.

Reference distribution	χ^2 distribution
Decision Rule (at the 1% Level)	$p < 0.01 \Rightarrow X_n$ non-random

References for Test

[1] Michael Baron and Andrew L. Rukhin. "Distribution of the number of visits of a random walk". *Stochastic Models* 15.3 (1999), 593–597

[2] Pál Révész. *Random walk in random and non-random environments*. World Scientific, 2005

[3] Frank Spitzer. *Principles of random walk*. Vol. 34. Springer, 2001

B.15 Random Excursions Variant Test

The focus of this test is the total number of times that a particular state is visited (i.e., occurs) in a cumulative sum random walk. The purpose of this test is to detect deviations from the expected number of visits to various states in the random walk. This test is actually a series of eighteen tests (and conclusions), one test and conclusion for each of the states: -9, -8, …, -1 and +1, +2, …, +9.

Reference distribution	half-normal distribution
Decision Rule (at the 1% Level)	$p < 0.01 \Rightarrow X_n$ non-random

References for Test

[1] Michael Baron and Andrew L. Rukhin. "Distribution of the number of visits of a random walk". *Stochastic Models* 15.3 (1999), 593–597

[2] Pál Révész. *Random walk in random and non-random environments*. World Scientific, 2005

[3] Frank Spitzer. *Principles of random walk*. Vol. 34. Springer, 2001

Bibliography

[1] A. Adya, W.J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J.R. Douceur, J. Howell, J.R. Lorch, M. Theimer, and R.P. Wattenhofer. "FARSITE: Federated, available, and reliable storage for an incompletely trusted environment". *ACM SIGOPS Operating Systems Review* 36 (2002), 1–14 (cited on p. 14)

[2] 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), 9 (cited on p. 87)

[3] N. Agrawal, V. Prabhakaran, T. Wobber, J.D. Davis, M. Manasse, and R. Panigrahy. "Design tradeoffs for SSD performance". *USENIX Annual Technical Conference, 2008*, 57–70 (cited on p. 120)

[4] S. Aiken, D. Grunwald, A.R. Pleszkun, and J. Willeke. "A performance analysis of the iSCSI protocol". *Proceedings of the 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST), 2003.* IEEE, 123–134 (cited on p. 13)

[5] S. Akyürek and K. Salem. "Adaptive block rearrangement". *ACM Transactions on Computer Systems (TOCS)* 13.2 (1995), 89–121 (cited on pp. 25, 109)

[6] S. Amarasinghe, D. Campbell, W. Carlson, A. Chien, W. Dally, E. Elnohazy, M. Hall, et al. *ExaScale Software Study: Software Challenges in Extreme Scale Systems*. Tech. rep. sponsored by DARPA IPTO in the context of the ExaScale Computing Study, 2010 (cited on p. 57)

[7] M. Arlitt, L. Cherkasova, J. Dilley, R. Friedrich, and T. Jin. “Evaluating content management techniques for web proxy caches”. *ACM SIGMETRICS Performance Evaluation Review* 27.4 (2000), 3–11 (cited on p. 113)

[8] E. Artiaga and A. Miranda. “PRACE-2IP Technical Deliverable D12.4: Performance Optimized Lustre”. *INFRA-2011-2.3.5 – Second Implementation Phase of the European High Performance Computing (HPC) service PRACE* (2012) (cited on p. 8)

[9] Fibre Channel Industry Association. *Fibre Channel Roadmaps - Retrieved 7/10/2013*. `http://www.fibrechannel.org/fibre-channel-roadmaps.html` (cited on p. 84)

[10] A. Azagury, V. Dreizin, M. Factor, E. Henis, D. Naor, Y. Rinetzky, O. Rodeh, J. Satran, A. Tavory, and L. Yerushalmi. “Towards an object store”. *Proceedings of the 20th IEEE Conference on Mass Storage Systems and Technologies (MSST), 2003*, 165–176 (cited on p. 32)

[11] Y. Azar, A.Z. Broder, A.R. Karlin, and E. Upfal. “Balanced allocations”. *SIAM Journal on Computing* 29.1 (1999), 180–200 (cited on p. 65)

[12] M. Baker, S. Asami, E. Deprit, J. Ouseterhout, and M. Seltzer. “Nonvolatile memory for fast, reliable file systems”. *ACM SIGPLAN Notices* 27.9 (1992), 10–22 (cited on p. 24)

[13] M.G. Baker, J.H. Hartman, M.D. Kupfer, K.W. Shirriff, and J.K. Ousterhout. “Measurements of a distributed file system”. *Proceedings of the thirteenth ACM symposium on Operating systems principles, 1991*. ACM, 198–212. isbn: 0897914473 (cited on pp. 26, 85)

[14] Jon Louis Bentley. “Solutions to Klee’s rectangle problems”. *Unpublished manuscript* (1977) (cited on p. 48)

[15] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, et al. “ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems” (2008) (cited on p. 11)

[16] M. Bhadkamkar, J. Guerra, L. Useche, S. Burnett, J. Liptak, R. Rangaswami, and V. Hristidis. “BORG: block-reORGanization for self-optimizing storage systems”. *Proceedings of the 7th conference on File and Storage Technologies (FAST), 2009*. USENIX Association, 183–196 (cited on pp. 25, 101, 109)

[17] M. Blaum, J. Brady, J. Bruck, and J. Menon. "EVENODD: An Optimal Scheme for Tolerating Double Disk Failures in RAID Architectures". *Proceedings of the 21st International Symposium on Computer Architecture (ISCA), 1994*, 245–254 (cited on p. 51)

[18] P.J. Braam. *The Lustre Storage Architecture*. Cluster File Systems Inc. architecture, design, and manual for Lustre. Nov. 2002. url: `http://www.lustre.org/docs/lustre.pdf` (cited on pp. 14, 19)

[19] R.P. Brent. "Uniform random number generators for supercomputers". *Proceedings of the Fifth Australian Supercomputer Conference, 1992, Melbourne*, 95–104 (cited on p. 67)

[20] A. Brinkmann and S. Effert. "Redundant Data Placement Strategies for Cluster Storage Environments". *Proceedings of the 12th International Conference on Principles of Distributed Systems (OPODIS), 2008* (cited on pp. 23, 36, 54)

[21] A. Brinkmann, S. Effert, F. Meyer auf der Heide, and C. Scheideler. "Dynamic and Redundant Data Placement". *Proceedings of the 27th IEEE International Conference on Distributed Computing Systems (ICDCS), 2007*. Toronto, Canada (cited on pp. 21, 23, 30, 35, 62)

[22] A. Brinkmann, M. Heidebuer, F. Meyer auf der Heide, U. Rückert, K. Salzwedel, and M. Vodisek. "V: Drive - Costs and Benefits of an Out-of-Band Storage Virtualization System". *Proceedings of the 21st IEEE Conference on Mass Storage Systems and Technologies (MSST), 2004*, 153–157 (cited on p. 31)

[23] A. Brinkmann, K. Salzwedel, and C. Scheideler. "Efficient, Distributed Data Placement Strategies for Storage Area Networks". *Proceedings of the 12th ACM Symposium on Parallel Algorithms and Architectures (SPAA), 2000*, 119–128 (cited on pp. 20, 40)

[24] A. Brinkmann, K. Salzwedel, and C. Scheideler. "Compact, adaptive placement schemes for non-uniform distribution requirements". *Proceedings of the 14th ACM Symposium on Parallel Algorithms and Architectures (SPAA), 2002*. Winnipeg, Manitoba, Canada, 53–62 (cited on pp. 21, 32, 34, 54)

[25] N. Brown. *Online RAID-5 resizing. drivers/md/raid5.c in the source code of Linux Kernel 2.6.18*. 2006 (cited on p. 108)

[26] J.S. Bucy, J. Schindler, S.W. Schlosser, and G.R. Ganger. “The DiskSim Simulation Environment Version 4.0 Reference Manual”. *Parallel Data Laboratory, Carnegie Mellon University* (2008), 26 (cited on p. 120)

[27] B. Callaghan, B. Pawlowski, and P. Staubach. *NFS version 3 protocol specification*. Tech. rep. RFC 1813, Network Working Group, 1995 (cited on p. 14)

[28] P. Cao and S. Irani. “Cost-aware WWW proxy caching algorithms”. *Proceedings of the 1997 USENIX Symposium on Internet Technology and Systems*. Vol. 193 (cited on p. 113)

[29] P.M. Chen and E.K. Lee. *Striping in a RAID level 5 disk array*. Vol. 23. 1. ACM, 1995 (cited on p. 120)

[30] P.M. Chen and D.A. Patterson. *Maximizing performance in a striped disk array*. Vol. 18. 3a. ACM, 1990 (cited on p. 107)

[31] Y. Chen, K. Srinivasan, G. Goodson, and R. Katz. “Design implications for enterprise storage systems via multi-dimensional trace analysis”. *Proceedings of the 23rd ACM Symposium on Operating Systems Principles, 2011*. ACM, 43–56 (cited on p. 27)

[32] W.S. Chou. “On inversive maximal period polynomials over finite fields”. *Applicable Algebra in Engineering, Communication and Computing* 6.4-5 (1995), 245–250 (cited on p. 69)

[33] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar. “Row-Diagonal Parity for Double Disk Failure Correction”. *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST), 2004*. San Francisco, CA, 1–14 (cited on p. 51)

[34] T. Cortes and J. Labarta. “Extending heterogeneity to RAID level 5”. *Proceedings of the USENIX Annual Technical Conference, 2001*. Boston, Massachusetts, 119–132 (cited on p. 20)

[35] T. Coughlin and E. Grochowski. “Years of Destiny: HDD Capital Spending and Technology Developments from 2012–2016” (June 19, 2012). URL: http://ewh.ieee.org/r6/scv/mag/MtgSum/Meeting2012_06_Presentation.pdf (cited on p. 3)

[36] B. Dawes, D. Abrahams, and R. Rivera. “Boost C++ libraries”. *Online at* http://www.boost.org (2009) (cited on p. 71)

[37] M. De Berg, O. Cheong, and M. Van Kreveld. *Computational geometry: algorithms and applications*. Springer, 2008 (cited on p. 48)

[38] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. "Dynamo: Amazon's Highly Available Key-value Store". *ACM SIGOPS Operating Systems Review* 41.6 (2007), 205–220 (cited on p. 22)

[39] A. Devulapalli, D. Dalessandro, and P. Wyckoff. "Data Structure Consistency Using Atomic Operations in Storage Devices". *Proceedings of the 5th International Workshop on Storage Network Architecture and Parallel I/Os (SNAPI), 2008*. Baltimore, USA, 65 –73 (cited on p. 32)

[40] J.R. Douceur and W.J. Bolosky. "A large-scale study of file-system contents". *Proceedings of the 1999 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. ACM, 59–70. isbn: 158113083X (cited on p. 87)

[41] D. Eastlake and P. Jones. *US secure hash algorithm 1 (SHA1)*. 2001 (cited on p. 44)

[42] D. Ellard, J. Ledlie, P. Malkani, and M. Seltzer. "Passive NFS tracing of email and research workloads". *Proceedings of the 2nd USENIX Conference on File and Storage Technologies, 2003*. USENIX Association, 203–216 (cited on pp. 26, 85, 86, 95, 103)

[43] M. Fasheh. "OCFS2: The Oracle Clustered File System, Version 2". *Ottawa Linux Symposium* (2006) (cited on p. 18)

[44] R. Freitas, J. Slember, W. Sawdon, and L. Chiu. "GPFS Scans 10 Billion Files in 43 Minutes". *IBM Advanced Storage Laborator. IBM Almaden Research Center. San Jose, CA* 95120 (2011) (cited on p. 3)

[45] Fujitsu Ltd. "Analyzing the Trends in the Enterprise Hard Disk Drive Industry" (2006). White paper. Online: `http://www.fujitsu.com/downloads/COMP/fcpa/hdd/enterprise-hdd-single_wp.pdf` (cited on p. 2)

[46] J. Gantz and D. Reinsel. "Extracting value from chaos". *IDC research report, Framingham, MA, June. Retrieved September* 19 (2011), 2011 (cited on p. 2)

[47] J. Gantz and D. Reinsel. "The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east". *IDC iView: IDC Analyze the Future* (2012) (cited on p. 2)

[48] Gartner Inc. *Solid-State Drives Will Complement, Not Replace, Hard-Disk Drives in Data Centers*. Apr. 15, 2013. url: `http://www.storagenewsletter.com/news/marketreport/gartner-ssd-hdd` (cited on p. 3)

[49] T.J. Gibson and E.L. Miller. "Long-term file activity and inter-reference patterns". *Computer Measurement Group (CMG 98) Proceedings* (1998) (cited on p. 26)

[50] A. Goel, C. Shahabi, S.Y.D. Yao, and R. Zimmermann. "SCADDAR: An efficient randomized technique to reorganize continuous media blocks". *Proceedings of the 18th International Conference on Data Engineering, 2002*. IEEE, 473–482 (cited on pp. 22, 108)

[51] M. Gómez and V. Santonja. "Characterizing temporal locality in I/O workload". *Proc. of the International Symposium on Performance Evaluation of Computer and Telecommunication Systems, 2002* (cited on pp. 101, 109)

[52] J.L. Gonzalez and T. Cortes. "Increasing the capacity of RAID5 by online gradual assimilation". *Proceedings of the international workshop on Storage network architecture and parallel I/Os, 2004*. ACM, 17–24 (cited on pp. 23, 30, 108)

[53] J.L. Gonzalez and T. Cortes. "Evaluating the effects of upgrading heterogeneous disk arrays". *International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS 2006)*. Calgary, Canada (cited on p. 30)

[54] J.L. Gonzalez and T. Cortes. "Distributing Orthogonal Redundancy on Adaptive Disk Arrays". *Proceedings of the International Conference on Grid computing, high-performAnce and Distributed Applications (GADA), 2008*. Monterrey, Mexico (cited on p. 20)

[55] M. Greenberger. "Method in randomness". *Communications of the ACM* 8.3 (1965), 177–179 (cited on p. 68)

[56] M. Haahr. "Random.org: True random number service". *School of Computer Science and Statistics, Trinity College, Dublin, Ireland. Website URL:* `http://www.random.org` (2010) (cited on pp. 65, 66)

[57] R.W. Hamming. "Error Detection and Error Correction Codes". *The Bell System Technical Journal* 26.2 (1950), 147–160 (cited on p. 16)

[58] R.W. Hamming. *Coding and information theory*. Prentice-Hall, Inc., 1986 (cited on p. 142)

[59] X. He, Q. Yang, and M. Zhang. "A caching strategy to improve iSCSI performance". *Proceedings of the 27th Annual IEEE Conference on Local Computer Networks, 2002*. IEEE, 278–285 (cited on p. 24)

[60] R.J. Honicky and E.L. Miller. "A fast algorithm for online placement and reorganization of replicated data". *Proceedings of the 17th IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2003*. Nice, France (cited on pp. 22, 36)

[61] R.J. Honicky and E.L. Miller. "Replication Under Scalable Hashing: A Family of Algorithms for Scalable Decentralized Data Distribution". *Proceedings of the 18th IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2004* (cited on pp. 22, 36)

[62] W.W. Hsu, A.J. Smith, and H.C. Young. "The automatic improvement of locality in storage systems". *ACM Transactions on Computer Systems (TOCS)* 23.4 (2005), 424–473 (cited on pp. 25, 109)

[63] J. D. Hunter. "Matplotlib: A 2D graphics environment". *Computing In Science & Engineering* 9.3 (2007), 90–95 (cited on p. 187)

[64] F. Hupfeld, T. Cortes, B. Kolbeck, J. Stender, E. Focht, M. Hess, J. Malo, J. Marti, and E. Cesario. "The XtreemFS architecture". *Proceedings of the LinuxTag 2007* (cited on p. 19)

[65] F. Hupfeld, T. Cortes, B. Kolbeck, J. Stender, E. Focht, M. Hess, J. Malo, J. Marti, and E. Cesario. "XtreemFS – a case for object-based storage in Grid data management". *Proceedings of 33th International Conference on Very Large Data Bases (VLDB) Workshops, 2007* (cited on p. 19)

[66] IBM Corp. "System 360 Scientific Subroutine Package Version II Programmer's Manual, H20-0205-1" (1967), 54 (cited on p. 68)

[67] G. Ifrah, E.F. Harding, D. Bellos, S. Wood, et al. *The Universal History of Computing: From the Abacus to Quantum Computing*. John Wiley & Sons, Inc., 2000 (cited on p. 1)

[68] S. Jin and A. Bestavros. "GreedyDual* Web caching algorithm: exploiting the two sources of temporal locality in Web request streams". *Computer Communications* 24.2 (2001), 174–183 (cited on p. 113)

[69] N. L. Johnson and S. Kotz. *Urn Models and Their Applications*. New York: John Wiley and Sons, 1977 (cited on p. 32)

[70] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. "Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web". *Proceedings of the 29th ACM Symposium on Theory of Computing (STOC), 1997*. El Paso, Texas, USA, 654–663 (cited on pp. 20, 30, 34, 61)

[71] D.R. Kenchammana-Hosekote, R.A. Golding, C. Fleiner, and O.A. Zaki. "The design and evaluation of network RAID protocols". *Research report RJ* 10316 (2004) (cited on p. 18)

[72] D.E. Knuth. "Volume 2: Seminumerical Algorithms". *The Art of Computer Programming* (1997), 192 (cited on pp. 68, 143)

[73] W. Koch and M. Schulte. *The Libgcrypt Reference Manual*. URL: `http://www.gnu.org/software/libgcrypt/` (cited on p. 71)

[74] I. Koltsidas and S.D. Viglas. "Flashing up the storage layer". *Proceedings of the VLDB Endowment* 1.1 (2008), 514–525 (cited on p. 24)

[75] S. Lee and H. Bahn. "Data allocation in MEMS-based mobile storage devices". *Consumer Electronics, IEEE Transactions on* 52.2 (2006), 472–476 (cited on p. 86)

[76] C.B. Legg. *Method of increasing the storage capacity of a level five RAID disk array by adding, in a single step, a new parity block and N–1 new data blocks which respectively reside in a new columns, where N is at least two*. US Patent 6,000,010. Dec. 1999 (cited on p. 23)

[77] D.H. Lehmer. "Mathematical methods in large-scale computing units". *Ann. Comput. Lab. Harvard Univ* 26 (1951), 141–146 (cited on p. 67)

[78] A.W. Leung, S. Pasupathy, G. Goodson, and E.L. Miller. "Measurement and analysis of large-scale network file system workloads". *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, 213–226 (cited on pp. 26, 87, 101)

[79] E. Levy and A. Silberschatz. "Distributed File Systems: Concepts and Examples". *ACM Computing Surveys* 22.4 (1990), 321–374 (cited on p. 11)

[80] D. Li and J. Wang. "EERAID: energy efficient redundant and inexpensive disk array". *Proceedings of the 11th workshop on ACM SIGOPS European workshop 2004*. ACM, 29 (cited on p. 120)

[81] Z. Li, Z. Chen, S.M. Srinivasan, and Y. Zhou. "C-miner: Mining block correlations in storage systems". *Proceedings of the 3rd USENIX Conference on File and Storage Technologies, 2004*. Vol. 186. USENIX Association (cited on p. 25)

[82] M. Luby. *Pseudorandomness and Cryptographic Applications*. Princeton University Press, 1996. isbn: 978-0691025469 (cited on p. 70)

[83] P. Lyman. "How much information? 2003". http://www.sims.berkeley.edu/research/projects/how-much-info-2003/ (2003) (cited on pp. 84, 116)

[84] G. Marsaglia. "Random numbers fall mainly in the planes". *Proceedings of the National Academy of Sciences of the United States of America* 61.1 (1968), 25 (cited on p. 68)

[85] G. Marsaglia and A. Zaman. "A new class of random number generators". *The Annals of Applied Probability* (1991), 462–480 (cited on p. 69)

[86] M. Matsumoto and T. Nishimura. "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator". *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 8.1 (1998), 3–30 (cited on p. 70)

[87] J. Matthews, S. Trika, D. Hensgen, R. Coulson, and K. Grimsrud. "Intel® Turbo Memory: Nonvolatile disk caches in the storage hierarchy of mainstream computer systems". *ACM Transactions on Storage (TOS)* 4.2 (2008), 4 (cited on p. 24)

[88] 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), 181–197 (cited on p. 107)

[89] N. Megiddo and D.S. Modha. "ARC: A self-tuning, low overhead replacement cache". *Proceedings of the 2nd USENIX Conference on File and Storage Technologies, 2003*, 115–130 (cited on p. 114)

[90] B. Mende, L. Noll, and S. Sisodiya. "How Lavarand Works". *Silicon Graphics Incorporated, published on Internet:* http://lavarand.sgi.com *(also reported in Scientific American)* (1997), 18 (cited on p. 66)

[91] M. Mense and C. Scheideler. "SPREAD: An Adaptive Scheme for Redundant and Fair Storage in Dynamic Heterogeneous Storage Systems". *Proc. of the 19th ACM-SIAM Symposium on Discrete Algorithms (SODA), 2008*. San Francisco, California (cited on p. 23)

[92] M. Mesnier, G. Ganger, and E. Riedel. "Object-based storage: pushing more functionality into storage". *Potentials, IEEE* 24.2 (2005), 31–34 (cited on p. 19)

[93] M. Mesnier, G.R. Ganger, and E. Riedel. "Object-based storage". *Communications Magazine, IEEE* 41.8 (2003), 84–90 (cited on p. 19)

[94] Microsoft Corporation. *Microsoft Windows ReadyBoost*. url: http://www.microsoft.com/windows/windows-vista/features/readyboost.aspx (cited on p. 24)

[95] A. Miranda and T. Cortes. "Analyzing Long-Term Access Locality to Find Ways to Improve Distributed Storage Systems". *20th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, 2012*. IEEE. Garching, Germany, 544–553. doi: 10.1109/PDP.2012.15 (cited on pp. 8, 101)

[96] A. Miranda and T. Cortes. "CRAID: Online RAID Upgrades Using Dynamic Hot Data Reorganization". *Proceedings of the 12th USENIX Conference on File and Storage Technologies, 2014*. Santa Clara, CA: USENIX, 133–146. isbn: 978-1-931971-08-9 (cited on p. 8)

[97] 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". *Proceedings of the 18th International Conference on High Performance Computing, 2011*. IEEE. Bangalore, India, 1–10. doi: 10.1109/HiPC.2011.6152745 (cited on p. 8)

[98] A. Miranda, S. Effert, Y. Kang, E.L. Miller, A. Brinkmann, and T. Cortes. *Data Distribution Simulator*. http://dadisi.sourceforge.net/. 2011 (cited on p. 44)

[99] A. Miranda, S. Effert, Y. Kang, I. Popov, T. Friedetzky, E.L. Miller, A. Brinkmann, and T. Cortes. "Random Slicing: Efficient and Scalable Data Placement for Large-scale Storage Systems". *ACM Transactions on Storage* 10.3 (2014), 36 (cited on p. 8)

[100] M.D. Mitzenmacher. “The Power of Two Choices in Randomized Load Balancing”. PhD thesis. Computer Science Department, University of California at Berkeley, 1996 (cited on p. 32)

[101] D. Narayanan, A. Donnelly, and A. Rowstron. “Write off-loading: Practical power management for enterprise storage”. *ACM Transactions on Storage (TOS)* 4.3 (2008), 10 (cited on p. 100)

[102] D. Narayanan, E. Thereska, A. Donnelly, S. Elnikety, and A. Rowstron. “Migrating server storage to SSDs: analysis of tradeoffs”. *Proceedings of the 4th ACM European conference on Computer systems, 2009*. ACM, 145–158 (cited on p. 120)

[103] T. Nightingale, Y. Hu, and Q. Yang. “The design and implementation of DCD device driver for UNIX”. *Proceedings of the 1999 USENIX Technical Conference*, 295–308 (cited on p. 24)

[104] B. Nowicki. “NFS: Network File System Protocol Specification”. *Network Working Group RFC* 1094 (1989) (cited on p. 18)

[105] J.K. Ousterhout, H. Da Costa, D. Harrison, J.A. Kunze, M. Kupfer, and J.G. Thompson. “A trace-driven analysis of the UNIX 4.2 BSD file system”. *Proceedings of the tenth ACM symposium on Operating systems principles, 1985*. ACM, 24. isbn: 0897911741 (cited on pp. 25, 85)

[106] J. Park, H. Chun, H. Bahn, and K. Koh. “G-MST: A dynamic group-based scheduling algorithm for MEMS-based mobile storage devices”. *Consumer Electronics, IEEE Transactions on* 55.2 (2009), 570–575 (cited on p. 86)

[107] S.K. Park and K.W. Miller. “Random number generators: good ones are hard to find”. *Communications of the ACM* 31.10 (1988), 1192–1201 (cited on p. 142)

[108] D. A. Patterson, G. Gibson, and R. H. Katz. “A Case for Redundant Arrays of Inexpensive Disks (RAID)”. *Proceedings of the 1988 ACM Conference on Management of Data (SIGMOD)*, 109–116 (cited on pp. 20, 51)

[109] D.A. Patterson, G. Gibson, and R.H. Katz. *A case for redundant arrays of inexpensive disks (RAID)*. Vol. 17. 3. ACM, 1988 (cited on p. 14)

[110] D.A. Patterson et al. “A simple way to estimate the cost of downtime”. *Proc. 16th Systems Administration Conf.| LISA 2002*, 185–8 (cited on p. 109)

[111] H. Payer, M.A. Sanvido, Z.Z. Bandic, and C.M. Kirsch. "Combo drive: Optimizing cost and performance in a heterogeneous storage device". *First Workshop on Integrating Solid-state Memory into the Storage Hierarchy, 2009*. Vol. 1. 1, 1–8 (cited on p. 24)

[112] PCMAG.COM. *Will Toshiba's Bit-Patterned Drives Change the HDD Landscape?* Aug. 19, 2010. URL: http://www.pcmag.com/article2/0,2817,2368023,00.asp (cited on p. 3)

[113] W.H. Press. *Numerical recipes in Fortran 77: the art of scientific computing*. Vol. 1. Cambridge university press, 1992 (cited on p. 67)

[114] M. Raab and A. Steger. ""Balls into Bins"–A Simple and Tight Analysis". *Randomization and Approximation Techniques in Computer Science* (1998), 159–170 (cited on pp. 32, 65)

[115] K.K. Ramakrishnan, P. Biswas, and R. Karedla. "Analysis of file I/O traces in commercial computing environments". *Proceedings of the 1992 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*. ACM, 78–90. ISBN: 0897915070 (cited on p. 26)

[116] H.J. Richter, A.Y. Dobin, O. Heinonen, K.Z. Gao, R.J. Veerdonk, R.T. Lynch, J. Xue, D. Weller, P. Asselin, M.F. Erden, et al. "Recording on Bit-Patterned Media at Densities of 1 Tb/in^2 and Beyond". *IEEE Transactions on Magnetics* 42.10 (2006), 2255–2260 (cited on p. 3)

[117] O. Rodeh and A. Teperman. "zFS: a scalable distributed file system using object disks". *Proceedings of the 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies, 2003*. IEEE Computer Society Washington, DC, USA, 207–218. ISBN: 0-7695-1914-8 (cited on p. 14)

[118] D. Roselli, J.R. Lorch, and T.E. Anderson. "A comparison of file system workloads". *Proceedings of the USENIX Annual Technical Conference, 2000*. USENIX Association, 4 (cited on pp. 26, 85, 87, 95, 103)

[119] R.E. Rottmayer, S. Batra, D. Buechel, W.A. Challener, J. Hohlfeld, Y. Kubota, L. Li, B. Lu, C. Mihalcea, K. Mountfield, et al. "Heat-assisted magnetic recording". *IEEE Transactions on Magnetics* 42.10 (2006), 2417–2421 (cited on p. 3)

[120] C. Ruemmler and J. Wilkes. *Disk shuffling*. Tech. rep. HPL-91-156, Hewlett Packard Laboratories, 1991 (cited on pp. 25, 109)

[121] C. Ruemmler and J. Wilkes. "UNIX disk access patterns". *Proceedings of the Winter 1993 USENIX Technical Conference*, 405–420 (cited on p. 109)

[122] A. Rukhin, J. Soto, J. Nechvatal, M. Smid, E. Barker, S. Leigh, M. Levenson, M. Vangel, D. Banks, A. Heckert, et al. *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*. Tech. rep. DTIC, 2001 (cited on pp. 67, 72)

[123] P. Sanders. "Reconciling Simplicity and Realism in Parallel Disk Models". *Proceedings of the 12th ACM-SIAM Symposium on Discrete Algorithms (SODA), 2001.* SIAM, Philadelphia, PA, 67–76 (cited on p. 21)

[124] M. Satyanarayanan. "Scalable, secure, and highly available distributed file access". *Computer* 23.5 (1990), 9–18 (cited on p. 18)

[125] M. Satyanarayanan, J.J. Kistler, P. Kumar, M.E. Okasaki, E.H. Siegel, and D.C. Steere. "Coda: a highly available file system for a distributed workstation environment". *IEEE Transactions on Computers* 39.4 (1990), 447–459 (cited on p. 18)

[126] C. Schindelhauer and G. Schomaker. "Weighted Distributed Hash Tables". *Proceedings of the 17th ACM Symposium on Parallel Algorithms and Architectures (SPAA), 2005.* Las Vegas, Nevada, USA, 218–227 (cited on p. 21)

[127] F. Schmuck and R. Haskin. "GPFS: A Shared-Disk File System for Large Computing Clusters". *Proceedings of the 1st USENIX Conference on File and Storage Technologies* (2002) (cited on p. 18)

[128] Seagate Technology PLC. *Seagate Cheetah 15K.5 FC product manual.* URL: `http://www.seagate.com/staticfiles/support/disc/manuals/enterprise/cheetah/15K.5/FC/100384772f.pdf` (cited on p. 120)

[129] F.L. Severence. *System modeling and simulation: an introduction.* John Wiley & Sons, 2009 (cited on p. 68)

[130] S. Shepler, M. Eisler, D. Robinson, B. Callaghan, R. Thurlow, D. Noveck, C. Beame, and N. Appliance. "Network file system (NFS) version 4 protocol". *Network Working Group RFC 3530* (2003) (cited on p. 14)

[131] A.J. Smith. "Analysis of Long Term File Reference Patterns for Application to File Migration Algorithms". *IEEE Transactions on Software Engineering* 7.4 (1981), 403–417. ISSN: 0098-5589 (cited on p. 25)

[132] G. Soundararajan, V. Prabhakaran, M. Balakrishnan, and T. Wobber. "Extending SSD lifetimes with disk-based write caches". *Proceedings of the 8th USENIX conference on File and storage technologies, 2010*. USENIX Association, 8–8 (cited on p. 24)

[133] M. Stevens, A.K. Lenstra, and B. de Weger. "Chosen-Prefix Collisions for MD5 and Colliding X.509 Certificates for Different Identities". *Proceedings of the 26th Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT) , 2007*, 1–22 (cited on p. 74)

[134] M. Stevens, A. Sotirov, J. Appelbaum, A.K. Lenstra, D. Molnar, D.A. Osvik, and B. de Weger. "Short Chosen-Prefix Collisions for MD5 and the Creation of a Rogue CA Certificate". *Proceedings of the 29th Annual International Cryptology Conference (CRYPTO), 2009*, 55–69 (cited on p. 74)

[135] I. Stoica, R. Morris, D. Liben-Nowell, D.R. Karger, M.F. Kaashoek, F. Dabek, and H. Balakrishnan. "Chord: a scalable peer-to-peer lookup protocol for internet applications". *IEEE/ACM Transactions on Networking* 11.1 (2003), 17–32 (cited on p. 31)

[136] J. Tate, F. Lucchese, R. Moore, and International Business Machines Corporation. International Technical Support Organization. *Introduction to storage area networks*. IBM, International Technical Support Organization, 2006 (cited on p. 13)

[137] M. Uysal, A. Merchant, and G.A. Alvarez. "Using MEMS-based storage in disk arrays" (2003) (cited on p. 24)

[138] A. Verma, R. Koller, L. Useche, and R. Rangaswami. "Srcmap: Energy proportional storage using dynamic consolidation". *Proceedings of the 8th USENIX Conference on File and Storage Technologies, 2010*. USENIX Association, 20–20 (cited on pp. 100, 101, 109)

[139] J. Viega. "Practical Random Number Generation in Software". *Proceedings of the 19th Annual Computer Security Applications Conference (ACSAC), 2003*, 129–141 (cited on p. 70)

[140] P. Vongsathorn and S.D. Carson. "A system for adaptive disk rearrangement". *Software: Practice and Experience* 20.3 (1990), 225–242 (cited on pp. 25, 109)

[141] J. Walker. "ENT Test suite". *Online:* http://www.fourmilab.ch/random (1998) (cited on p. 72)

[142] J. Walker. "HotBits: Genuine random numbers, generated by radioactive decay". *Online at* http://www.fourmilab.ch/hotbits (2001) (cited on p. 66)

[143] T. Wang. "Integer hash function". *Online:* http://www.concentric.net/~ttwang/tech/inthash.htm (2007) (cited on p. 44)

[144] X. Wang, Y.L. Yin, and H. Yu. "Finding Collisions in the Full SHA-1". *Proceedings of the 25th Annual International Cryptology Conference (CRYPTO), 2005*, 17–36 (cited on p. 74)

[145] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn. "CRUSH: Controlled, Scalable And Decentralized Placement Of Replicated Data". *Proceedings of the ACM/IEEE Conference on Supercomputing, 2006*. Tampa, FL (cited on p. 22)

[146] S.A. Weil, S.A. Brandt, E.L. Miller, D.D.E. Long, and C. Maltzahn. "Ceph: A Scalable, High-Performance Distributed File System". *OSDI'06: Proceedings of the 7th Symposium on Operating System Design and Implementation*. Seattle, WA, USA, 307–320 (cited on pp. 14, 19, 31)

[147] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou. "Scalable Performance of the Panasas Parallel File System". *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)*, 17–33 (cited on p. 19)

[148] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. "The HP AutoRAID hierarchical storage system". *ACM Transactions on Computer Systems (TOCS)* 14.1 (1996), 108–136 (cited on p. 24)

[149] C.K. Wong. "Minimizing expected head movement in one-dimensional and two-dimensional mass storage systems". *ACM Computing Surveys (CSUR)* 12.2 (1980), 167–178 (cited on p. 25)

[150] T.M. Wong, G.R. Ganger, J. Wilkes, et al. *My Cache Or Yours?: Making Storage More Exclusive*. School of Computer Science, Carnegie Mellon University, 2000 (cited on p. 86)

[151] C. Wu and X. He. "GSR: A Global Stripe-based Redistribution Approach to Accelerate RAID-5 Scaling". *Parallel Processing (ICPP), 2012 41st International Conference on*. IEEE, 460–469 (cited on pp. 23, 108)

[152] Q. Yang and Y. Hu. "DCD—disk caching disk: A new approach for boosting I/O performance". *23rd Annual International Symposium on Computer Architecture, 1996*. IEEE, 169–169 (cited on p. 24)

[153] G. Zhang, J. Shu, W. Xue, and W. Zheng. "SLAS: An efficient approach to scaling round-robin striped volumes". *ACM Transactions on Storage (TOS)* 3.1 (2007), 3 (cited on p. 108)

[154] W. Zheng and G. Zhang. "FastScale: Accelerate RAID Scaling by Minimizing Data Migration". *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST), 2011* (cited on pp. 21, 23, 120)

[155] R.M. Ziff. "Four-tap shift-register-sequence random-number generators". *Computers in Physics* 12 (1998), 385 (cited on p. 69)

Further Reading

[1] E. Anderson, M. Arlitt, C.B. Morrey III, and A. Veitch. "DataSeries: an efficient, flexible data format for structured serial data". *ACM SIGOPS Operating Systems Review* 43.1 (2009), 70–75. issn: 0163-5980

[2] E. Anderson, S. Spence, R. Swaminathan, M. Kallahalla, and Q. Wang. "Quickly finding near-optimal storage designs". *ACM Transactions on Computer Systems (TOCS)* 23.4 (2005), 337–374. issn: 0734-2071

[3] L. Carroll. *Alice in wonderland*. Vol. 836. Pelangi Publishing Group Bhd, 1942

[4] P.M. Chen, E.K. Lee, G.A. Gibson, R.H. Katz, and D.A. Patterson. "RAID: High-performance, reliable secondary storage". *ACM Computing Surveys (CSUR)* 26.2 (1994), 145–185

[5] T. Cortes and J. Labarta. "HRaid: A Flexible Storage-system Simulator". *Proceedings of the International Conference on parallel and Distributed Processing Techniques and Applications (PDPTA'99)*. Las Vegas, NV

[6] T. Cortes and J. Labarta. *A Case for Heterogeneous Disk Arrays*. Tech. rep. Departament d'Arquitectura de Computadors - Universitat Politècnica de Catalunya, 2000

[7] A.B. Downey. "The structural cause of file size distributions". *Proceedings of the 9th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, 2001.* IEEE, 361–370. isbn: 0769513158

[8] M. Factor, K. Meth, D. Naor, O. Rodeh, and J. Satran. "Object storage: The future building block for storage systems". *Local to Global Data Interoperability-Challenges and Technologies, 2005*. IEEE, 119–123

[9] S. Ghemawat, H. Gobioff, and S. Leung. "The Google File System". *SOSP'03: Proceedings of the nineteenth ACM symposium on Operating systems principles*. New York, NY, USA: ACM, 29–43. isbn: 1-58113-757-5

[10] K. Goda and M. Kitsuregawa. "The history of storage systems". *Proceedings of the IEEE* 100.13 (2012), 1433–1440

[11] J. Guerra, H. Pucha, J. Glider, W. Belluomini, and R. Rangaswami. "Cost effective storage using extent based dynamic tiering". *Proceedings of the 9th USENIX conference on File and storage technologies, 2011.* USENIX Association, 20–20

[12] J.H. Howard and Carnegie-Mellon University. Information Technology Center. *An overview of the andrew file system*. Carnegie Mellon University, Information Technology Center, 1988

[13] S. Kavalanekar, B. Worthington, Q. Zhang, and V. Sharda. "Characterization of storage workload traces from production Windows servers". *IEEE International Symposium on Workload Characterization, 2008. IISWC 2008.* IEEE, 119–128

[14] G.W.F. von Leibniz. *Monadologie*. Braumüller und Seidel, 1847

[15] D.T. Meyer and W.J. Bolosky. "A study of practical deduplication". *ACM Transactions on Storage (TOS)* 7.4 (2012), 14

[16] D.A. Muntz, P. Honeyman, and C.J. Antonelli. "Evaluating delayed write in a multilevel caching file system". *Proceedings of the IFIP/IEEE International Conference on Distributed Platforms: Client/Server and Beyond: DCE, CORBA, ODP and Advanced Distributed Applications, 2002*. IEEE, 415–429. isbn: 0412732807

[17] D. Narayanan, A. Donnelly, E. Thereska, S. Elnikety, and A. Rowstron. "Everest: Scaling down peak loads through I/O off-loading". *Proceedings of the 8th USENIX conference on Operating systems design and implementation, 2008*. USENIX Association, 15–28

[18] J. von Neumann. "Various techniques used in connection with random digits". *Applied Math Series* 12.36-38 (1951), 1

[19] I. Popov, A. Brinkmann, and T. Friedetzky. "On the Influence of PRNGs on Data Distribution". *20th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), 2012*. IEEE, 536–543

[20] T. Pratchett. *Interesting times*. HarperCollins, 2009

[21] P. Scheuermann, G. Weikum, and P. Zabback. "Data partitioning and load balancing in parallel disk systems". *The VLDB Journal* 7.1 (1998), 48–66

[22] A. Traeger, E. Zadok, N. Joukov, and C.P. Wright. "A nine year study of file system and storage benchmarking". *ACM Transactions on Storage (TOS)* 4.2 (2008), 1–56. issn: 1553-3077

[23] A.M. Turing. "Computing machinery and intelligence". *Mind* 59.236 (1950), 433–460

[24] W. Vogels. "File system usage in Windows NT 4.0". *Proceedings of the seventeenth ACM symposium on Operating systems principles, 1999*. ACM, 93–109. isbn: 1581131402

[25] J. Wilkes. "Traveling to Rome: QoS specifications for automated storage system management". *Quality of Service—IWQoS 2001* (2001), 75–91

[26] R. Yellin. "The data storage evolution. Has disk capacity outgrown its usefulness?" *Terada magazine* (2006)

[27] Q. Zhu, Z. Chen, L. Tan, Y. Zhou, K. Keeton, and J. Wilkes. "Hibernator: helping disk arrays sleep through the winter". *ACM SIGOPS Operating Systems Review, 2005*. Vol. 39. 5. ACM, 177–190

Index

A

access patterns (data blocks)
(see working sets)
access sequentiality
(see working sets)
AFS, 18
ALIS, 25
ARC, 113
atmospheric noise, 66
(see also TRNGs)

B

BORG, 25

C

C-Miner, 25
CAGR, 2, 3
Ceph, 14
(see also SAN filesystems, OSDs)
CODA, 18
coefficient of variation, 127
compound annual growth rate
(see CAGR)
Consistent Hashing, 34
adaptivity, 61–62
fairness, 53, 55
memory usage, 57
performance, 57–60
CRAID, 4, 110
architecture, 110–112
archive partition, 110
cache partition, 110
control flow, 111
I/O Monitor, 112–114
I/O Redirector, 115
Mapping Cache, 114–115
response time, with parities,
122–127
response time, without parities,
129–131
workload balancing, with parities,
127–129
workload distribution, without
parities, 131–132
CRAID-0, 119
CRAID-0$^+$, 119

CRAID-5, 116
CRAID-5^{+}, 116
CRAID-5_{d}, 117
CRAID-5_{d}^{+}, 117
cryptographic hash functions, 70

D

DAS architectures, 13
data layout optimizations, 25
(see also C-Miner, ALIS, BORG)
data rebalancing, 83–84
data storage
annual growth, 2
digital data online, 2
history, 1
direct-attached storage
(see DAS architectures)
DiskSim, 120
distributed file systems, 12
block-based, 18
(see also NFS, AFS, CODA, OCFS2)
clients, 12
file servers, 12
metadata, 12
object-based, 19
(see also OSDs, MDSs, Panasas, Lustre, Ceph, XtreemFS, GoogleFS)
parallel, 18
(see also OCFS2, GPFS, Panasas)
storage servers, 12

E

ecuyer1988, 79
ENT statisctical test, 72
EvenOdd-Codes, 51

F

Farsite, 14
(see also SAN filesystems)
Fibre Channel, 13
file systems, 11
file operations, 12
files, 11

G

GDSF, 113
generalized feedback shift register generators
(see also PRNGs types)
generalized feedback shift registers, 69–70
GPFS, 18
GSR, 108

H

hard disk drive
(see HDDs)
HDD technology trends, 2–3
areal density, 2
bit-patterned recording, 3
CPP/GMR heads, 3
heat assisted recording, 3
rotational latency, 2
seek time, 2
shingled recording, 3
HDDs
(see also HDD technology trends)
hellekalek1995, 79
heterogeneous RAID

(see RAID)
hierarchical storage, 24
AutoRAID, 24
DCD, 24
HDD+SSD storage, 24
(see also Turbo Memory, ReadyBoost, Griffin, Combo Drive)
iCache, 24
MEMS, 24
NVRAM (battery backed), 24
hypergeometric distribution, 36 (*n.*)

I

information storage
(see data storage)
inversive congruential generators, 68–69
(see also PRNGs types)
iSCSI, 13
(see also SCSI, Fibre Channel)

K

kreutzer1986, 79

L

lagged Fibonacci generators, 69
(see also PRNGs types)
lagged_fibonacci607, 79
lags
(see lagged Fibonacci generators)
large-scale storage systems
extensibility, 4
heterogeneity, 4
minimum rebalancing, 5
rebalancing, 4
redundancy, 30
scalability issues, 5
lava lamps, 66
(see also TRNGs)
LBA, 19
lifespan (data blocks)
(see working set)
linear congruential generators, 67–68
(see also PRNGs types)
linear feedback shift register generators
(see also PRNGs types)
linear feedback shift registers, 70
logical block address
(see LBA)
LRU, 112
Lustre, 14, 19
(see also SAN filesystems, OSDs)

M

MD5, 79
MDSs, 19
Mersenne twister generator, 70
(see also generalized feedback shift registers)
minstd_rand, 79
minstd_rand0, 79
mt11213b, 79
mt19937, 79
multiplicative congruential generators, 68
(see also PRNGs types)

N

NAS architectures, 13
network-attached storage

(see NAS architectures)
NFS, 14, 18
(see also NAS architectures)
NIST statisctical test, 72

O

object-based storage, 18–19
metadata servers
(see MDSs)
object storage devices
(see OSDs)
OSDs, 19
storage object, 19
(see also distributed file systems)
OCFS2, 18

P

Panasas, 19
partitioning, benefits of, 110–111
PRNG, 72
PRNGS
influence on distribution fairness, 72–77
PRNGs, 65–66
efficiency, 66
influence on distribution performance, 77–80
period, 67
seed, 67
sequence uniformity, 66
types, 67–70
pseudo-random number generators
(see PRNGs)

R

radioactive decay, 66
(see also TRNGs)
RAID, 14
error correction, 14
failure probability, 15
hamming codes, 16
heterogeneity, 17
mirroring, 14
rebalancing, 23
FastScale, 23
gradual assimilation, 23
GSR, 23
MDM, 23
restriping, 108
small writes, 17
striping, 14
(see also RAID standard levels)
RAID standard levels, 15–17
(see also RAID)
RAID-0, 117
RAID-5, 116
RAID-5+, 116
rand48, 79
Random Slicing, 4, 37–43, 119
adaptivity, 62
data lookup, 41
extensions, 43
fairness, 55
fault tolerance, 41
interval creation algorithm, 38, 40–41
CutShift, 40
CutShift+Sorted, 41
Greedy, 40
Greedy+Sorted, 41
intervals, 37
memory usage, 60
parity support, 117
performance, 60

scalability, 44–48
segment tree, 48
randomized data distribution, 20–23
balls into bins, 32
adaptivity, 33
capacity efficiency, 33
compactness, 33
competitiveness, 33
fairness, 33
time efficiency, 33
Consistent Hashing, 20
CRUSH, 22
cut-and-paste, 20
Dynamo, 22
hash function, 20
(see also PRNGs)
redundancy group, 32
Redundant Share, 22
relative capacity, 32
replication, 21
RUSH, 22
SCADDAR, 22
Share, 21
Sieve, 21
Spread, 23
trivial replication, 35
randomness, 65–66
connection with fairness, 65
ranlux3, 79
ranlux3_01, 79
ranlux64_3, 79
ranlux64_3_01, 79
Redundant Share, 35–36
adaptivity, 62
fairness, 54–55
memory usage, 60
performance, 60
Reed-Solomon codes, 51
RUSH, 36–37
adaptivity, 62
fairness, 54–57
memory usage, 60
performance, 60

S

SAN architectures, 13
SAN filesystems, 14
SAS, 13
SATA, 4
SCSI, 4, 13
Seagate Cheetah, 120
SEQ, 117
SHA1, 79
Share, 34–35
fairness, 53–54
simulator
CRAID, 120–121
data distribution, 43
solid-state device
(see SSDs)
SSD, 12, 24
SSDs, 3, 4
manufacturing capacity, 3
storage architectures, 13
(see also DAS architectures,
NAS architectures,
SAN architectures,
SAN filesystems)
storage area networks
(see SAN architectures)
striping
(see RAID)
subtract with carry generators, 69
(see also PRNGs types)

T

taus88, 79
Tiger-192, 79

(see also SAN filesystems, OSDs)
traces
animation collection, 86
dbs, 86
nfsc, 86
nfss, 86
render, 86
vcs, 86
cello99, 85–86
deasna, 86
home02, 86–87
TRNGs, 66
true random number generators
(see TRNGs)

U

Unix_rand, 79

W

WLRU, 114
working sets
access patterns (data blocks), 95
access sequentiality, 96–98
density of access, 98–99
lifespan (data blocks), 93–95
overlap, 87
shared blocks, 88–92

X

XtreemFS, 19

Z

zFS, 14

List of Acronyms

ARC	Adaptive Replacement Cache
BCE	Before the Common/Current/Christian Era
CAGR	Compound Annual Growth Rate
CDF	Cumulative Distribution Function
CE	Common/Current/Christian Era
CIFS	Common Internet File System
CPP/GMR	Current Perpendicular to Plane/Giant Magnetoresistance
DAS	Direct-Attached Storage
GDSF	Greedy-Dual-Size with Frequency
GFSR	Generalized Feedback Shift Register
HDD	Hard Disk Drive
ICG	Inverse Congruential Generator
iSCSI	Internet Small Computer System Interface
LBA	Logical Block Address
LCG	Linear Congruential Generator
LFG	Lagged Fibonacci Generator
LFSR	Linear Feedback Shift Register

LFUDA	Least-Frequently Used with Dynamic Aging
LRU	Least-Recently Used
MCG	Multiplicative Congruential Generator
MD5	Message Digest #5
MDS	Metadata Server
MEMS	Microelectromechanical Systems
NAS	Network-Attached Storage
NFS	Network File System
NIST	National Institute of Standards and Technology
NSD	Network Storage Device
NVRAM	Non-volatile RAM
OSD	Object Storage Device
P_A	Archive partition
P_C	Cache partition
PRNG	Pseudo-Random Number Generator
QoS	Quality of Service
RAID	Redundant Array of Independent Disks
RAM	Random-access Memory
SAN	Storage Area Network
SAS	Serial Attached SCSI
SATA	Serial Advanced Technology Attachment
SCSI	Small Computer System Interface
SHA1	Secure Hash Algorithm #1
SLA	Service-Level Agreement
SSD	Solid-State Device
SWCG	Subtract With Carry Generator
TRNG	True Random Number Generator
WLRU	Weighted Least-Recently Used

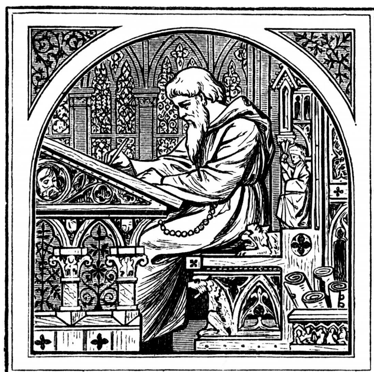
Colophon

This thesis, which includes 41 figures and 13 tables, was typeset using the LuaLaTeX typesetting system originally developed by Jonathan Kew, based on TeX created by Donald Knuth.

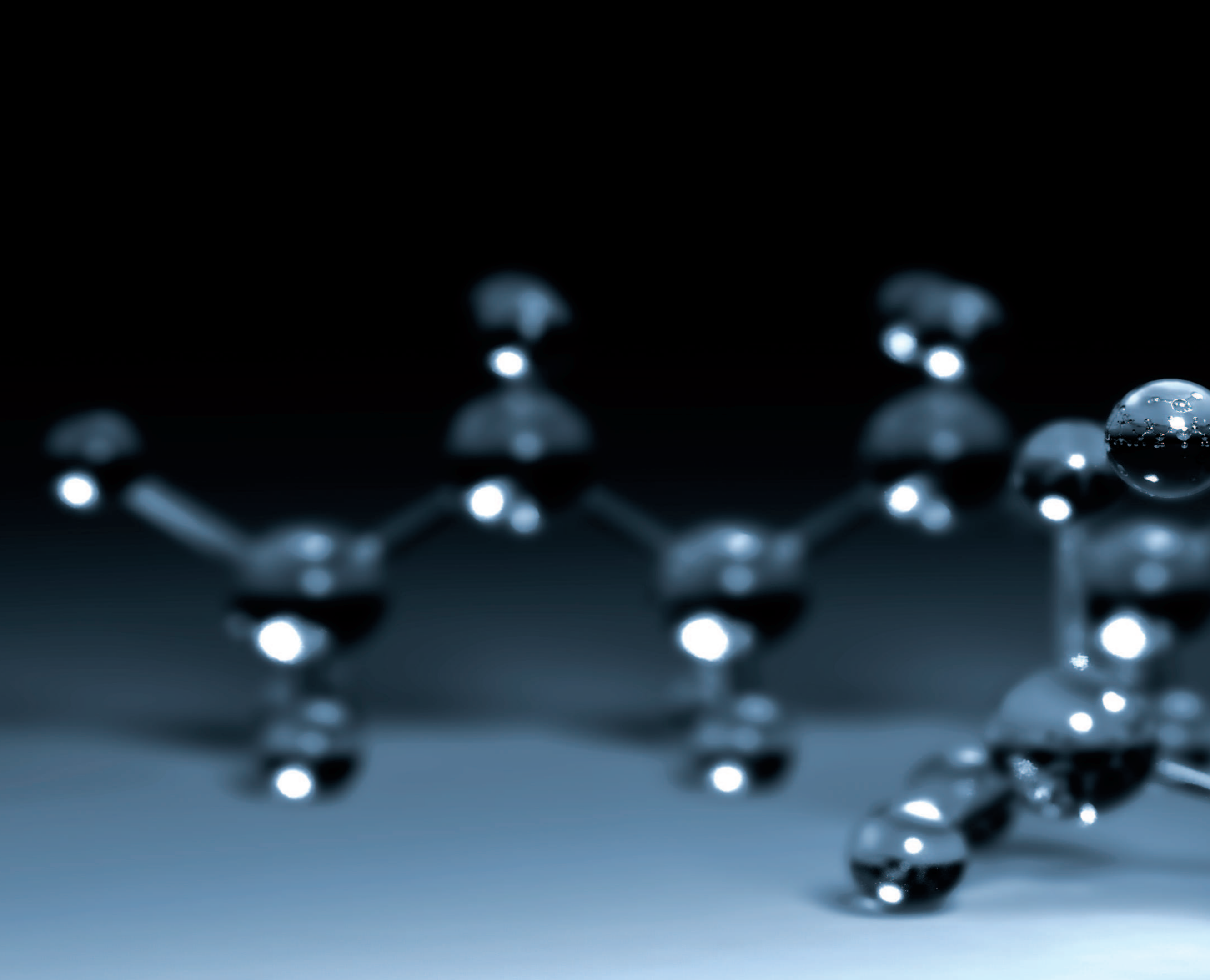
The body text is set 11/13.6pt on a 29pc measure with Minion Pro designed by Robert Slimbach. Other fonts include Adobe's Myriad Pro and TheSansMono Condensed by Luc(as) de Groot. Mathematical equations are typeset using XITS Math and Asana Math. The title page uses League Gothic and Adobe Caslon Pro.

The use of sidenotes instead of footnotes was inspired by Edward Tufte's *Beautiful Evidence*.

Figures were prepared using matplotlib [63], PGF/TikZ and, to a lesser extent, gnuplot.



© 2009–2014 by Alberto Miranda Bueno.
All rights reserved.



The evolution of computer systems has brought an exponential growth in the volumes of data which is pushing the capabilities of current storage to organize and access this information effectively: as the creation and demand for computer-generated data grows at an estimated rate of 40% to 60% per year, large storage architectures need increasingly scalable data layouts that are able to adapt to this growth and provide the sufficient performance, since inadequate layouts may lead to unbalancing problems like bottlenecks or resource underutilization.

This dissertation presents the results of our research in scalable data layouts that can adapt to continuously increasing volumes of data. With this thesis, we make several novel contributions to storage research: first we design and evaluate a pseudorandom distribution strategy that can adapt to hardware changes with minimal rebalancing; second, we undertake a comparative study about the influence of pseudorandom number generators on the performance and distribution quality of several randomized data layouts; third, we conduct an analysis of long-term data access patterns in several real-world traces in order to determine if it is possible to offer high performance and a balanced load with less than minimal data rebalancing; fourth, we use the knowledge learned on long-term access patterns to design an extensible RAID architecture that can adapt to changes in the number of disks without migrating large amounts of data.