UNDERSTANDING THE FAULT-TOLERANCE PROPERTIES OF LARGE-SCALE STORAGE SYSTEMS

BY

ERIC WILLIAM DAVIS ROZIER

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2011

Urbana, Illinois

Doctoral Committee:

Professor William H. Sanders, Chair and Director of Research
Professor Gul Agha
Professor Stephen Levinson
Professor Mahesh Viswanathan
Dr. Pin Zhou, IBM Almaden Research Center

ABSTRACT

Modern storage systems continue to increase in scale and complexity as they attempt to meet the increasing storage needs of our society. Additionally, increased requirements to comply with government regulation and consumer expectations have increased the need to make data more available and reliable for longer periods of time. The design of modern and next-generation storage systems is a difficult task that requires high storage capacity and efficiency while also maintaining the data integrity.

The rapid advancement of storage system technologies brings with it a level of uncertainty as to the fitness of new designs and methods for meeting the complex requirements. New technologies, like deduplication, promise improved storage efficiency, but their impact on reliability measures is unclear due to the complex relationships inherent to the systems that employ these technologies. Additionally, as systems scale up, they become subject to faults and errors that previous-generation systems may never have encountered due to the rare nature of these faults. Because of the stiffness of the represented systems, and the complex relationships involved, it can be difficult to analyze these environments correctly and efficiently.

In this dissertation, we propose a method to analyze storage system reliability by using component-based models coupled with realistic fault models. We solve these complex systems by identifying fault, fault propagation, and mitigation events; by identifying dependence relationships between state variables, events, and rewards; and by decomposing our model at various points during model solution to improve the efficiency of our solution while maintaining the correctness of our reward measures. In particular, we discuss building scalable component-based models of large-scale systems that employ modern reliability methods, such as RAID, and state-of-the-art storage efficiency methods such as dedupli-

cation. We present detailed fault models for these systems, including a novel model for undetected disk errors. To enable efficient solution of these models we propose a method to analyze the dependence relationships that underlie storage systems and propose a way to solve these models by identifying and exploiting these relationships when solving for reliability measures. We apply our methods to real-world systems, detail the consequences for the reliability of deduplication, and suggest and evaluate methods to improve reliability while still maintaining improved storage efficiency.

To my wife, parents, and brother for all of their love and support.

ACKNOWLEDGMENTS

I would first like to thank my family for all of their love, support, and encouragement. My wife, Kristin, has been with me every step of the way as we both pursued our degrees, even though we have been thousands of miles apart for most of the process. I thank my parents for their support of my academic pursuits throughout my life and nurturing of my interests. I thank my brother Stuart for sharing so much of my life and being a great friend in addition to being a great brother.

I would like to thank the many teachers that influence my career as a student, and in particular to those who helped point my future towards graduate studies in research. I am grateful to Mrs. Martin, who thought me how think and reason about the world in unique ways; Mrs. Wingfield who taught me the fundamentals of science and research; and Dr. Ciardo, Dr. Henson, and Dr. Smirni who gave me the opportunity to begin my research career at William and Mary.

I would also like to thank my PERFORM family. I am grateful to my advisor, Professor William H. Sanders, for his technical and financial support, and for providing me with the academic freedom to pursue my interests regardless of our funding. I am thankful to Jenny Applequist for the education in grammar she has given me, the appreciation she has instilled in me for the serial comma, and the many long hours of work she has graciously spent helping me with my writing. I would also like to thank Professor Sanders for somehow selecting such a strong core of like-minded students for our group. All of them have been mentors and supporters for my Ph.D., and close friends: Shravan Gaonkar, Ryan Lefever, Sankalp Singh, Adnan Agbaria, Robin Berthier, Shuyi Chen, Tod Courtney, David Daly, Nathan Dautenhahn, Salem Derisavi, Doug Eskins, Michael Ford, Mark Griffith, Michael Ihde, Kaustubh Joshi, Ken Keefe, Vinh Lam, Elizabeth LeMay, James Lyons, Michael Mc-

Quinn, Hari Ramasamy, and Saman Aliari Zonouz.

I am indebted to the researchers of the IBM Almaden Research Center's Storage Group for working with me on my thesis, for their support through my fellowship, and for their co-authorship on the papers I have written. My dissertation would not have been possible without the guidance and hard work of Wendy Belloumini, Jim Hafner, K. K. Rao, Veera Deenadhayalan, Nagapramod Mandagere, Sandeep Uttamchandani, Mark Yakushev, and Pin Zhou. I would like to add extra thanks to Pin for her support and mentoring, both when I was interning at IBM and when I was working at Illinois. It was a great opportunity to have her as an unofficial co-advisor, as I expanded the systems aspects of my work.

I thank Professors Sanders, Agha, Levinson, Viswanathan, and Dr. Zhou for serving on my committee and offering me their support, guidance, and suggestions. Their technical feedback during my preliminary examination was critical to my thesis, and their support throughout this process helped me to shape my research into a successful dissertation.

I would also like to extend my gratitude to Pioneer Hi-Bred, France Telecom, Rockwell Collins, and IBM, whose financial support during my graduate career made this research possible.

Lastly, but not least, I would like to thank God, and the supportive and loving community of family, friends, and mentors He has provided to me for my graduate studies.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

CFS	Cluster File System
CTMC	Continuous-Time Markov Chain
DFA	Deterministic Finite Automaton
DTMC	Discrete-Time Markov Chain
ECC	Error Correction Code
HPC	High-Performance Computing
LSE	Latent Sector Error
MDG	Model Dependency Graph
MTTDL	Mean Time to Data Loss
MTTF	Mean Time to Failure
NFA	Non-deterministic Finite Automaton
RAID	Redundant Array of Inexpensive Disks
SAN	Storage Area Network
UDE	Undetected Disk Error
URE	Undetected Read Error
UWE	Undetected Write Error

LIST OF SYMBOLS

M	A model represented by a 5-tuple $(S, E, \Phi, \Lambda, \Delta)$
S	A finite set of state variables that take values from the set of natural numbers
E	A finite set of events that may occur in a model
Φ	An event-enabling function specification that maps a set of state variable assignments to a Boolean value
Λ	A transition rate function specification that maps a set of state variable assignments to the set $(0, \infty)$
Δ	A state variable transition function specification that maps a set of state variable assignments to a set of state variable assignments
Ψ	The state of a model as determined by its state variables
$\mathcal{R}$	A rate reward structure
$\mathcal{I}$	An impulse reward structure
$\mathcal{P}(S, \mathbb{N})$	A set of partial functions between S and $\mathbb{N}$
Θ_M	A set of reward variables for a model M
θ_i	A reward variable

CHAPTER 1

INTRODUCTION

Modern storage systems have become increasingly large, both because of the growth of user data, and the advent of new regulations concerning the length of time data must be kept. As storage systems have increased in size, failures have become increasingly common, and faults which once were rare enough to be ignored as improbable have become threats to data integrity. As systems and needs evolve, so do reliability goals, and technologies that once provided suitable reliability may need to be re-examined. To better understand the challenges and requirements of evolving storage systems before production and deployment, and to understand the costs and trade offs necessitated by alternative system configurations, methods must be adapted to evaluate and assess proposed designs.

Part of the process of evaluating proposed storage systems is the development of a complete understanding of the environment in which the system operates, interactions within the system, and the nature of the technologies employed by the system to achieve reliability goals and storage efficiency goals. Recent years have seen rapid development along these lines, especially in response to recent legislation mandating the length of time data must be stored for retrieval. In 2002, over five exabytes of data were produced [2], representing an increase of 30% from 2001. By 2007 the figure had increased to 281 exabytes, 10% more than expected because of faster growth in cameras, digital TV shipments, and other media sectors [1]. In 2010, the total data produced passed the zettabyte barrier. Forecasts put the total size of stored data for 2011 at 10 times the size for 2006, roughly 1.8 zettabytes [3]. Storing the data has become increasingly problematic. In 2007, as forecast, the amount of data created exceeded available storage for the first time [1].

Part of the driving force behind those changes has been the more than 10,000 legal regulations enacted in the U.S. alone [4]. In 2003, the amount of data stored for compliance

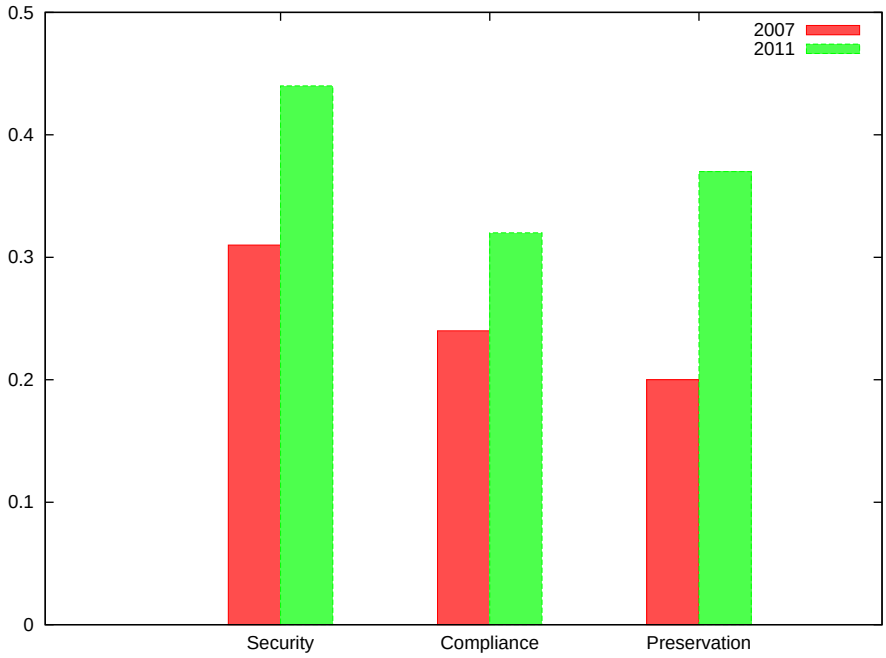


Figure 1.1: Percent of total data that requires high reliability due to security, compliance, or preservation requirements [1].

reasons increased by 63% [5]. What makes those increases particularly significant is that they took place before the most demanding regulations, such as Sarbanes-Oxley, went into effect. Furthermore, digital storage is also being impacted by the shift to digital storage of permanent records. As shown in Figure 1.1, an increasing amount of data is subject to high reliability requirements due to security, compliance, or preservation-related goals.

It is not enough, however, to simply have a large, reliable storage system. Despite the decreasing cost of COTS storage devices, the cost of managed disk-based storage is still high. These costs usually represent more than just the cost of the physical hardware itself. Enterprise-class storage has proven to be too expensive for many uses, creating an environment in which less expensive near-line drives are used for archival purposes [6].

Additional pressure on storage systems is being created by the expanding availability and reliance on cloud-based applications and resources. Web services such as e-mail, archives, photo sharing, and social networks require that large volumes of user data be stored indefinitely. In order for these services to remain economically viable to their operators, they must have access to low-cost storage that remains reliable and available for their customers.

To enable increased scalability for primary and archival storage systems new methods are being developed to improve storage efficiency, allowing a larger data set to be stored in less space by eliminating redundancy. Due to the increasing size requirements of archival storage, many system administrators are choosing to deploy these increased-efficiency systems without fully understanding the reliability consequences.

The key challenges addressed by this dissertation are as follows:

- Developing a robust understanding of current storage systems currently deployed, and being developed. This includes an understanding of the underlying system models, including the current storage requirements, technologies for improving storage efficiency and reliability, and the faults these systems face in production systems and next-generation systems currently being designed.

- Creating new fault models for emerging faults faced by modern and next-generation storage systems based on the analysis of real data from production systems, and the previous literature. This includes models of fault interaction, and mitigation techniques so that a rich fault environment can be appropriately simulated.

- Analyzing and developing new models for storage data, its placement on production file systems, and how the characteristics of that data can be used to better understand the impact of system designs, and to develop new techniques to improve reliability and storage efficiency of storage systems under development to enable intelligent design decisions.

- Creation of new techniques to improve the efficiency and correctness of our analysis, specifically aimed at confronting the challenges presented by the rare nature of certain important faults in modern storage systems.

- The development of new reliability infrastructures designed to provide storage systems which meet specified reliability goals while maintaining a high level of storage efficiency.

1.1 Thesis Statement

There exists a need for methods that allow for the simulation of complex storage systems with complex dependence relationships, accounting for wide classes of faults in order to design and plan for next-generation systems to meet goals for storage system reliability and efficiency.

It is our thesis that by combining component level models of storage system hardware, with empirically derived models of data dependence relationships, we can efficiently calculate measures of reliability for the systems in a rich environment of faults, and provide real solutions for achieving both reliability and storage efficiency goals.

To achieve this goal we must:

- Develop an understanding of the faults affecting not only the current generation of storage systems, but also those that will be important in next-generation systems.
- Develop an understanding of the technologies which will be employed by next-generation systems, such as deduplication.
- Understand the complex dependence relationships in modern storage systems which arise from RAID, and deduplication.
- Exploit the structure of the underlying models to improve efficiency of solution of stiff models, given the wide range of rates implied by large-scale systems with rare-errors and fast I/O.
- Illustrate the use of the developed methods on a real system, using component-based methods.
- Propose methods for improving the reliability of large-scale storage systems, while maintaining some improvements to storage-efficiency.

1.2 Related Work

We now discuss work related to this dissertation. While we will discuss related work more specific to our methods in each chapter to provide the context for the contributions of each chapter, we discuss here some work of general interest to our motivation.

1.2.1 Modeling Disk Faults

Magnetic storage have become remarkably reliable devices since their inception, despite their complexity. Both near-line and enterprise drives often implement the ability to detect and recover from many types of errors automatically. Faulty portions of a platter can be remapped to different logical and physical portions of the disk, and disks can often correctly report when their data is no longer readable. Manufacturers provide reliability parameters for their disks, such as mean time to failure (MTTF), and many studies exist which detail the failure rates of drives in practice [7, 8, 9]. In addition to traditional, whole disk failure, work has been done in the literature detailing other types of failures which result in the loss of individual sectors [10, 9].

While these fault models characterize many types of faults, they do not appear to be sufficient to estimate the reliability of large scale systems. IBM/LLNL's ASCI White with only 8192 cores, realized a MTTF of only 40 hours [11], much lower than expected from traditional fault models. While RAID was sufficient in the past to tolerate faults [12], faults which are unaddressed by RAID have been documented [13]. To assess the reliability of modern storage systems, more detailed models of observed faults are required.

1.2.2 The Impact of Modern Storage Technologies on Reliability

In order to reduce the footprint of backup and archival storage, system architects have begun using a new method to improve storage efficiency, called data deduplication. At a high level, data deduplication is a method for eliminating redundant data in a storage system to improve storage efficiency. Sub-file segments are fingerprinted and compared to a data base of identified segments to find duplicate data. Duplicate data is then replaced with

references to the stored instances. Recently this technology has also become available for near-line primary storage controllers. These same methods are also being used to diminish the silos between primary and archival storage for scale-out file systems. With the increasing prominence of cloud computing, providers are also actively evaluating deduplication as a method to decrease costs when supplying cloud storage products to customers.

The literature provides a good understanding on the cost of deduplication in terms of performance [14, 15]. Reliability studies have been much fewer in number. Since traditional deduplication keeps only a single instance of redundant data, deduplication has the potential to magnify the negative impact of losing a data chunk in chunk-based deduplication [16, 5] that divides a file into multiple chunks, or of missing a file in deduplication using delta encoding [17, 18, 19, 20, 21] that stores the differences among files. However, due to the smaller number of disks required to store deduplicated data, it also has the potential to improve reliability as well. Administrators and system architects have found understanding the data reliability of their system under deduplication to be important but extremely difficult [22]. A related question is estimating if a data reliability constraint for files associated with a business critical application (such as database) will be violated if data deduplication is employed.

Existing literature on the reliability of deduplication is hindered by a reliance on heuristics – the key recommendation is to keep multiple copies of a data chunk instead of storing only a single instance. Deep Store [5] proposed to determine the level of redundancy for a chunk based on a user-assigned value of importance. D. Bhagwat et al. [16] suggested that the number of replicas for a chunk should be proportional to its reference count, i.e., the number of files sharing the chunk. A gap exists in the current literature on the topic of quantifying the data reliability of a deduplication system or providing a means to estimate whether a set of reliability requirements can be met in a deduplication system.

1.3 Contributions

The contributions of this dissertation are as follows:

- A model for an emerging class of faults, undetected disk errors (UDEs), derived from real data and an analysis of how UDEs interact with existing reliability methods which cause them to be both orthogonal to these methods, and to interact to propagate faults to the parity section of RAID groups.

- A model of data deduplication, analyzing the dependence relationships implied by data deduplication in a real system, and the complex ways in which deduplication can affect both reliability and storage efficiency of the entire disk, and file categories on the disk itself. We show via our model and analysis that such category specific information is important both to understand the impact of deduplication, but also to form efficient strategies for improving reliability of the underlying storage system without completely compromising the storage efficiency benefits.

- A method for analyzing dependence relationships in storage system models. This includes a method to represent the system model as a Model Dependency Graph (MDG), a data structure which encodes all direct and indirect dependence relationships as paths through the graph traversing state variables and events.

- A method for simulating systems with constructed MDGs. This new method attempts to improve performance by decomposing the underlying model based on the current values of all state variables, and the information encoded in the MDG, splitting the model into separated sub-models which can be analyzed independently until the next rare event.

- Experimental results to show the use of our methods on real production systems and data, demonstrating their utility for analyzing real world systems.

- A case study of a production archival storage system to show how reliability goals can be defined, system fitness for these goals can be evaluated, and how system models can be altered to meet the stated reliability goals.

- Novel reliability frameworks which can be used with deduplicated storage systems to provide additional reliability while increasing storage efficiency as well as methods to

protect against faults which were orthogonal to previous reliability methods.

1.4 Organization

The remainder of this dissertation is organized as follows.

Chapter 3 details the important faults facing modern and next-generation storage systems. We detail models for traditional disk failure, latent sector errors (LSEs), and develop new models for UDEs. We study the ways in which these faults can impact storage systems, the reasons the occur in real physical systems, and how they manifest when they occur. In addition to studying the faults themselves we address the way these faults are mitigated in production systems, and develop models of both the detection and recovery phases of mitigation for each of our fault models.

Chapter 2 details the creation of models for storage systems. In particular it introduces a component based modeling method which focuses on building, validating, and using models of individual components to enable scalability analysis, while giving some confidence in models of systems have not yet been built. We analyze an existing system in production at the National Center for Supercomputing Applications (NCSA) for our analysis.

Chapter 4 details the creation of fault models specific to our studied system, and introduces the idea of decomposition as a method for handling rare events and the stiffness which results from their introduction to our models.

Chapter 5 introduces the concept of a MDG. We begin by discussing the ways in which dependencies can occur within a model, their importance to our decomposition methods, and an algorithm for creating an MDG from a model. We then discuss the decomposition procedure, beginning with an analysis of the types of rare events which can change the dependence relationships represented in the MDG. We detail a method for modifying the MDG to represent the current dependence relationships and define a decomposed set of sub-models. We then discuss how to use these techniques to solve a model, and the limitations of our solution method.

In Chapter 6 we study a real deduplicated storage system, enabling us to develop a model of deduplication, expanding the models developed in Chapters 2 and 4. We present the results

of this study and develop a full model for an entire production system, solving the model and analyzing the impact of deduplication, and various proposed strategies for reliability using the techniques presented in 5.

Finally, in Chapter 7 we discuss the consequences of our studies for real systems and formally present novel methods for improving the reliability of primary and secondary storage systems, including the ability to tolerate UDEs, faults that previously had no acceptable means for tolerance.

CHAPTER 2

COMPONENT-BASED MODELS

2.1 Introduction

In this chapter we address the problem of building large scalable models of systems. One challenge that we tackle is that of developing models that can be trusted to adequately represent next-generation systems that have not yet been built. We approach that goal by developing component-based models based on current systems, validating these models against log data for current systems, and then constructing larger-scale models using these components. We present such an approach in this chapter, showing the results of scaling a component-based model of the Abe cluster [23] to petascale. We will use this method for component-based modeling in Chapters 4 and 6 when building models of next-generation systems.

Our results in this chapter emphasize the importance of developing a detailed model of the underlying system based on real data. The failure rates we observed in the real system differed from those presented in the manufacturer data sheets. Software faults due to the corrupted applications or the Lustre file system were responsible for many errors and had a repair time that was significantly less than hardware errors. Without detailed study of the underlying system, these important factors would have been unaccounted for, resulting in an incorrect analysis of our next-generation system.

2.2 Motivation

Historically, scientific computing has driven large-scale computing resources to their limits. In the last decade, plans to achieve petascale computing have come to fruition. While

supercomputer performance has improved by over two orders of magnitude every decade, the performance gap between the individual nodes and the overall processing ability of an entire system has widened drastically [24]. This has led to a shift in the paradigm of supercomputer design, from a centralized approach to a distributed one that supports heterogeneity. While most high-performance computing environments require parallel file systems, there have been several file systems, such as GPFS [25], PVFS2 [26], and Lustre [27], that have been specifically proposed to support very large-scale scientific computing environments.

As the number of individual computing resources and components becomes very large, the frequency of failure of components within these clusters and the propagation of these failures to other resources are important concerns to high-performance computing applications. Failures can be caused by many factors: (a) transient hardware faults due to increased chip density, (b) software error propagation due to a large buggy legacy code base, or (c) manufacturing defects and environmental factors such as temperature or humidity.

Recent literature on failure analysis of BlueGene/L discusses various causes of increased downtime of supercomputers [28]. It has been well-established that elimination of failures is impossible; it is only feasible to circumvent failures and to mitigate their effects on a system's performance. The standard approach to the mitigation of a failure is to checkpoint the application at regular intervals. Long et al., however, showed that check-pointing has a large impact on the performance of very large computer clusters with large numbers of nodes [29].

Increasing the number of compute servers in a cluster almost always increases the size of the desired storage subsystem. Depending on the type of parallel file system, that means an increase in the number of file servers that could accept requests from the compute servers to keep up with I/O requests. Compute servers and file servers have very different characteristics. First, a failure in a file server needs more attention than a failure in a compute node. A compute server might just be marked as unavailable until it is repaired, but a failed file server might have to be reconstructed, or its state might need to be transferred to another file server, depending on the replication strategy. Second, file servers are inherently slower than compute servers due to their I/O characteristics. This generally makes file servers the bottleneck for the reliability and performance of a cluster. Unfortunately, there has been a

trend towards increasing failure rates for I/O subsystems that is similar to that for overall petascale clusters. This increase in failures can be attributed to the increase in the number of individual components that are used to build the whole I/O subsystem . Recent studies have shown that workload intensity is highly correlated to the failure rates [30, 31]. That emphasizes the need for thorough analysis to understand the impact of the I/O subsystems and their failures on petascale computers.

To address the research challenge of providing realistic prediction of petascale file system availability, we took a two-pronged approach. First, we have obtained the failure event log of the Abe cluster from the NCSA. The log contains the failures of individual nodes, file server nodes, and the storage area network (SAN). We preprocessed the event logs to determine various reward measures of interest corresponding to the file system, such as the availability of the file system over the lifetime of the log and the failure rate of jobs due to I/O failures and other transient failures. Then, we built and refined stochastic models of the file system used by these clusters that abstracts much of the operations, while generating reward measures that are comparable to the real log events. We then scaled the models to reflect the scale and magnitude of a future petascale computer and estimated the impact of current I/O and file system designs on a petascale computer. Furthermore, we evaluated strategies that could be used to mitigate the bottlenecks due to scaling of I/O file system and cluster designs from current supercomputers to petascale computers. Our analysis will give storage architects support to make informed design choices as they build larger cluster file systems.

The rest of the Chapter is organized as follows. Section 2.3 outlines related work. Section 2.4 discusses the file system architecture of the Abe cluster at NCSA with the analysis of collected failure log files. Section 2.5 presents the conceptual stochastic activity network model of the Abe cluster. Section 2.6 covers results and analysis. Section 2.7 offers our conclusions and plans for future work.

2.3 Related Work

The estimation and prediction of the failure of file systems are crucial to understanding the overall performance of petascale computers. Past literature describes several attempts to model and analyze different aspects of large-scale supercomputing systems.

Log/trace-based analysis: Recent literature using trace-based system analysis has shown that storage subsystems are prone to higher failure rates than their makers estimate because of underrepresented disk infant mortality rates [32]. In addition to disk failures, the analysis by Jiang et al. shows that interconnects and protocol stacks play a significant role in storage subsystem failure [33]. Furthermore, Liang et al. investigated the failure events from the event logs from BlueGene/L to develop failure prediction models to anticipate future fatal failures [28]. In general, trace-based analysis of logs provide good metrics for evaluating and understanding working systems, but is limited to the scope of events represented by these traces, making it difficult to study trends or behaviors not witnessed in the traces.

Model-based analysis: Wang et al. looked at the impact on system performance in the presence of correlated failures as the systems are scaled to several hundred thousand processors [29]. Rosti et al. presented a formal model to capture CPU and I/O interactions in scientific applications, to characterize system performance [34].

The use of simulation for evaluating the model provides the ability to predict behavior and bottlenecks of future designs, but the accuracy of predictions may often be compromised by assumptions of parameter values. Our approach focuses on integrating trace based analysis with model-based evaluation to form a combined approach, providing guidance to make informed choices for system design. Using failure data from the logs of the cluster as parameter values, we verify our models against the real system. We then analyze the impact of current design choices when the system is scaled. Our approach reduces the burden of sensitivity analysis, reducing the design space to a moderate size providing the opportunity to perform a robust analysis of the system.

2.4 Abe Cluster: System Configuration and Log File Analysis

The Abe cluster architecture was the current state-of-the-art as of 2008. Abe consists of 1200 blade compute nodes, i.e., 9600 core CPU Intel 64 (2.33 GHz dual-socket quad-core) processors, 8/16 GB shared RAM per node, and an InfiniBand (IB) interface. The cluster runs Red Hat Enterprise Linux 4 (Linux 2.6.9) as its operating system. The cluster can provide a peak compute performance of 89.47 peta-FLOPS (floating point operations per second). The Lustre file system supports a 100TB parallel cluster file system for the Abe cluster's compute nodes [23].

2.4.1 General Cluster File System (CFS) Architecture

A typical storage architecture for a cluster file system consists of a *metadata server*, multiple *file servers*, and *clients* [27]. The metadata server maintains the file system's metadata, which includes the access control information, mapping of files and directory names to their locations, and mapping of allocated and free space. The metadata server serves the metadata to the clients. The file servers maintain the actual data and information about the file blocks stored on the connected I/O disks and serve these file blocks to the clients. For reliability/performance, the file blocks can be replicated/striped over multiple disks. The client communicates first with the metadata server and then with the appropriate file server to perform the required read and write operation. The readers are referred to [27] for further details.

2.4.2 Abe CFS Server Hardware

The Abe Lustre-FS is currently supported by 24 Dell dual Xeon servers that provide 12 fail-over pairs[1]. One OSS serves the metadata of the Lustre-FS, 8 OSSes serve the /cfs/scratch OSS, and the remaining 6 servers handle the remaining partitions of the shared file systems (home, local, usr, etc.) of the cluster. Each server self-monitors its file system's health. The 2 metadata OSSes are connected to the storage I/O through a dual 2Gb fiber channel (FC).

[1]We refer to a fail-over pair as an *OSS* in the remainder of the Chapter.

2.4.3 Abe CFS Storage Hardware

Scratch partition: 2 S2A9550 storage units from DataDirect Networks Systems provide the storage hardware for the CFS's scratch partition. Each S2A9550 supports 8 4Gb FC ports. Each port connects to 3 tiers of SATA disks. Each tier has (8+2) disks in a RAID6 configuration. Therefore, there are 480 disks, each with a 250GB capacity, that form the scratch partition providing 96TB of usable space.

Metadata: DDN EF2800 provides the I/O hardware to support the metadata of the Lustre-FS. It is connected to the 2 metadata OSSes through a dual 2Gb fiber channel. The EF2800 has one tier of 10 disks in RAID10 configuration.

Other partitions: 10 IBM DS4500s serve an approximate total of 40TB of usable space over a SAN via a 2Gb FC.

Lustre settings: Lustre version 1.4.10.X runs on all of the OSS's hardware. Most of the reliability is provided by the SAN hardware; therefore, the Lustre reliability features are switched off.

2.4.4 Abe Log Failure Analysis

All NCSA clusters have elaborate logging and monitoring services built into them. The log data set used in this study was collected from 05/03/2007 to 10/02/2007 for compute nodes (compute-logs) and from 09/05/2007 to 11/30/2007 for the SAN (SAN-logs). The compute-logs and SAN-logs are monitored precisely, and the logs provide details about the events taking place in the cluster. Events are reported with the node IP addresses and the event times appended to the log information. To extract accurate failure event information, we filter failure logs based on temporal and causal relationships between events.

Table 2.2 provides the availability of the Abe cluster based on the notifications provided by the SAN administrators to the users [35]. The availability of Abe's SAN can be estimated to be between 0.97 and 0.98 depending on the dates one chooses as the start and end times for the measure computation. Table 2.1 shows Lustre-FS mount failures experienced by individual compute nodes aggregated on a per-day basis. Lustre-FS mount failures do not always imply the failure of the CFS, as these errors could be caused by intermittent

network unavailability. Nevertheless, those errors are perceived as failures from the cluster's perspective.

Table 2.3 presents the job failure/completion statistics obtained by analyzing the compute-log. The analysis shows that the transient errors causing network unavailability (between the compute nodes and the CFS or between the compute nodes and the login nodes) are 5 times more likely to cause job failures than other errors are (such as software errors or CFS failures). Earlier clusters had dedicated back-planes connected to compute nodes to provide communication. Current communication in Abe is through COTS network ports and switches. The change in the design choice was motivated chiefly by a desire to lower costs and increase flexibility in maintaining the system.

Table 2.4 provides the disk failure and replacement log from 09/05/2007 to 11/28/2007 for disks that support the scratch partition of the Abe's cluster. The authors of [32] estimated the disks' hazard rate function to be statistically equivalent to a Weibull distribution. We performed similar survival analysis on the disk failure data and found that Weibull with $\beta = 0.7$ was a good fit for Abe's disk drive failure logs. The key insights we gained from analyzing failure data and from discussions with cluster system administrators are as follows:

- The disk replication redundancy and replacement have been so well-streamlined that they almost never cause catastrophic failure of the CFS. On average, 0-2 disks are replaced on the Abe cluster per week.

- The Abe cluster's S2A9550 RAID6 (8+2) technology combines the virtues of RAID3, RAID5, and RAID0 to provide both reliability and performance [36]. RAID6 prevents a second drive failure from occurring during disk re-mirroring. The *Blue Waters* peta-scale computer, which will be built at the University of Illinois, will likely have an

Date	#	Date	#	Date	#
07/03/07	102	07/19/07	258	08/16/07	375
08/20/07	591	09/05/07	005	09/17/07	002
09/18/07	004	09/19/07	003	09/28/07	463
09/29/07	477	10/01/07	051	10/02/07	035

Table 2.1: Lustre mount failure notification by compute nodes from 07/01/07 to 10/02/07; column with "#" represents the number of compute nodes that experienced mount failure

Lustre-FS outage time			
Cause of Failure	Start time	End time	Hours
I/O hardware	07/21/07 23:03	07/22/07 12:00	12.95
I/O hardware	07/31/07 01:49	07/31/07 20:01	18.18
I/O hardware	08/22/07 18:08	08/23/07 02:15	08.12
I/O hardware	08/28/07 16:20	08/29/07 18:01	01.67
I/O hardware	09/25/07 18:00	09/26/07 09:30	15.50
I/O hardware	10/04/07 09:30	10/04/07 21:55	12.42
Batch system	10/16/07 17:56	10/16/07 21:24	03.47
Network	10/29/07 11:53	10/29/07 15:15	03.36
File system	11/16/07 09:30	11/16/07 10:00	00.40
File system	11/19/07 09:04	11/19/07 11:00	01.93

Table 2.2: User notification of outage of the Lustre-FS

Total jobs submitted between 05/13/07 and 10/02/07	44085
Total failures due to transient network errors	1234
Total failures due to other/file system errors	0184

Table 2.3: Job execution statistics for the Abe cluster

(8+3) RAID configuration. That would make the failure of the file system due to multiple individual disk failures highly unlikely.

- Most file system failures are due to software errors, configuration errors, and other transient errors. The software errors take, on average, 2-4 hours to resolve. Most often, the fix is to bring the disks to a consistent state using a file system check (*fsck*). A hardware failure due to a network component or a RAID controller might take up to 24 hours to resolve, as these components need to be procured from a vendor.

2.5 Stochastic Activity Network Model: Cluster File System

The failure data analysis and the insights provide the details necessary to build a stochastic model of the Abe's cluster file system. Here, we describe the details of the stochastic activity network models using Möbius [38].

Dates in September 2007	05	06	09	13	23
Number of failed disks	2	1	1	1	1
Dates in October 2007	08	17	24		
Number of failed disks	2	1	1		
Dates in November 2007	08	17			
Number of failed disks	1	1			

Survival analysis of the disk failures ($n = 480$) using Weibull regression (in log relative-hazard form) gives the shape parameter as 0.6963571 with standard deviation of 0.1923109 (95% confidence interval) [37]

Table 2.4: Disk failure log from 09/05/2007 to 11/28/2007 for disks supporting the Abe cluster's scratch partition

2.5.1 Overall Model

Figure 2.1 shows the *composed model* of the Abe cluster using replicate/join composition in Möbius . The leaf nodes in the replicate/join tree are stochastic activity network models that implement the functionalities. Space limitations do not permit detailed descriptions of these submodels. The CLUSTER model has two main submodels connected using a join, where the models share states on error propagation from their CLIENT to the CFS. The CLIENT represents the behavior and interaction of the compute nodes and the communication network between the compute nodes and the CFS. The CFS_UNIT emulates the Abe's cluster file system. It is composed of the OSS, OSS_SAN_NW, SAN, and the DDN_UNITS. The OSS implements the availability and operational model of the metadata server and the file server. The OSS_SAN_NW implements the failure model of the network ports and switches that connect OSS to the DDN_UNITS. The SAN emulates the operations provided by the network to communicate between OSS and the DDN_UNITS. The OSS, OSS_SAN_NW, SAN, and the DDN_UNITS communicate by sharing information about their current state of operation and availability. The DDN_UNITS composes multiple RAID6_UNITS with RAID_CONTROLLER. The failure of disks in RAID6_UNITS is assumed to follow a Weibull distribution. RAID_CONTROLLER emulates the failure and operation of a typical RAID6 architecture. The DDN_UNITS is replicated to emulate multiple S2A9550 units.

Since the goal is to investigate the impact of availability of file systems to peta-scale computers, the stochastic activity network models do not consider hardware failure in compute nodes. Our model incorporates only the behavior of the scratch partition and the meta-

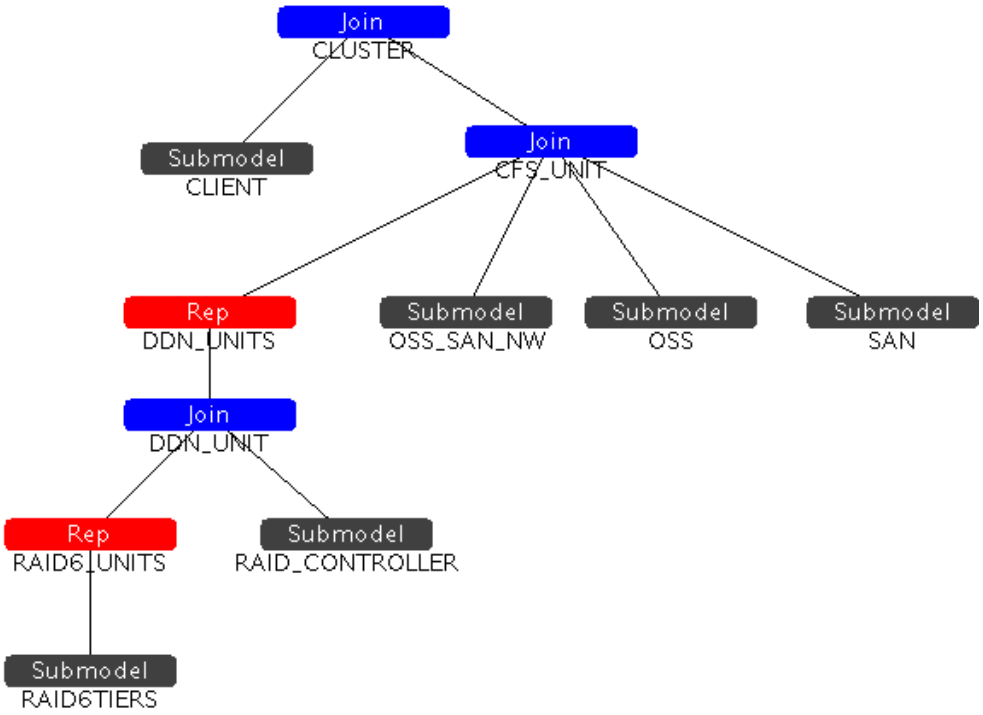


Figure 2.1: Compositional model of the CFS.

data servers of the CFS, because a cluster's utility depends mainly on its scratch partition's availability. Finally, hardware and software misconfiguration errors occur in the early deployment phase of the system; therefore, we exclude them from the models. In the following subsections, we describe the reward measures and the failure model used to represent the Abe's CFS.

2.5.2 Reward Measures

The *availability* of the cluster file system is defined as the ability of the CFS to serve the client nodes. More precisely, it is defined as the fraction of time when all the file server nodes (OSSes), the DDN, and the network interconnect between the OSSes and the DDN are in the *working* state.

The *disk replacement rate* is defined as the number of disks that need to be replaced per unit of time to sustain the maximum availability of the CFS.

The *cluster utility*, *CU*, is an availability metric from the cluster's perspective. To be precise, it is defined as $CU = \left(1 - \frac{Compute\ cycles\ lost\ due\ to\ unavailable\ file\ system}{Total\ available\ compute\ cycles}\right)$. CU is a metric that is different from the availability metric of the CFS[2]. The cluster users and SAN administrators tend to notice different levels of availability. The reasons are failures in network communication between the compute nodes and the CFS as well as failures due to intermittent transient errors that make CFS appear unavailable even though it has not failed.

2.5.3 Failure Model for Abe's CFS

The Abe's cluster suffers from failures mainly because of 3 types of errors: hardware errors, software errors, and transient errors. Each kind of error affects all the CFS's components.

The *hardware errors* in the metadata/file servers (OSSes) occur in the hardware components that are built to operate the system. These errors include processor, memory, and network errors. Hardware errors are assumed to be less frequent than disk failures, occurring

[2]CU does not distinguish between compute cycles used to perform check-pointing and those used for actual computation.

Model parameter	Values (range)
Disk MTBF[2]	100000-3000000
Annualized Failure Rate (AFR)	0.40%–8.6%
Weibull distribution's shape parameter[1]	0.6–1.0
Number of DDN[1]	2–20
Number of compute nodes[1]	1200–32000
Average time to replace disks[3]	1–12 hours
Average time to replace hardware[3]	12–36 hours
Average time to fix software[3]	2–6 hours
Job request per hour[1]	12–15 per hour
Hardware failure rate[1]	1–2 per 720 hours
Software failure rate	1–2 per 720 hours
Annual growth rate of disk capacity[2]	33%
DDN_Units[1]	2–20
OSS Units[1]	8–80

Parameter values obtained from: log file analysis [1], data specification from literature and hardware white papers [2], discussions with NCSA cluster administrators[3]

Table 2.5: Abe cluster's simulation model parameters

at the rate of 1–2 per month. The RAID controllers in the DDN or network ports/switches that connect DDN to OSS show similar failure rates. The repairs of these components take 12–36 hours depending upon the severity of the failure (as reported by SAN administrators), as the needed replacement parts have to be shipped from the vendors. Most of the hardware is replicated with fail-over mechanisms. Failure of both members of the fail-over pair causes the unavailability of the CFS system. The replacement of failed disks is modeled as a deterministic event. The repair time is varied from 1 to 12 hours across simulation experiments.

The *software errors* that cause failure of the cluster file systems are mainly due to the corrupted supercomputing applications running on the compute nodes (implemented in the CLIENT submodel) or the Lustre-FS (implemented in the OSS submodel). Since we do not have accurate estimates on software corruption errors, we assume that the rates are similar in the orders of magnitude to hardware error rates. The repair times for software errors are modeled as deterministic events. The repair time is varied from 2 to 6 hours across simulation experiments.

Transient errors occur in most components of the cluster model, but mainly in the network components. The error rates are obtained from the failure-log analysis as shown in Table 2.3.

Transient errors are temporary, but hard to diagnose. Our model assume that one of these errors causes a few minutes of unavailability of components under transient failure. The jobs depending on those components fail due to the temporary unavailability.

Past literature has emphasized the importance of modeling *correlated failures* [31]. Most correlated errors occur because of shared resources. Correlated errors propagate to components that have causal or spatial proximity. In the CFS model, hardware errors propagate because other hardware components are connected to each other. Software errors propagate from compute nodes to OSS or from OSS to disk, leading to data corruption. Transient errors propagate errors into software. All failures except disk failures are modeled as exponential distributions. To model correlated failures, we model jobs and requests submitted to the CFS, and estimate the probability p that the job requires a resource that is inaccessible due to failure, causing errors to propagate through the system.

2.6 Experimental Results and Analysis

We evaluate the design of the Abe cluster's availability using simulation in the Möbius tool. Table 2.5 summarizes the parameters collected through failure log analysis, hardware reliability specifications, and discussions with cluster administrators. In order to reflect the size and scale of a peta-scale computer and to determine the factors that impede the high availability of the CFS, we scale the number of individual components in the composed model. By implementing scaling through the addition of components (each of which has its own individual failure models) rather than by changing failure parameter values themselves, we ensure that the failure rates observed in the overall system model accurately reflect the new system size. All the simulation results are reported at a 95% confidence level.

2.6.1 Impact of Disk Failures on CFS

To evaluate the baseline effect of failures of disks on availability of the CFS, we evaluate the DDN_UNITS models associated with the RAID6 tiers and the RAID controllers in isolation from failures of other components of the SAN. Figure 2.2 shows the availability of the storage

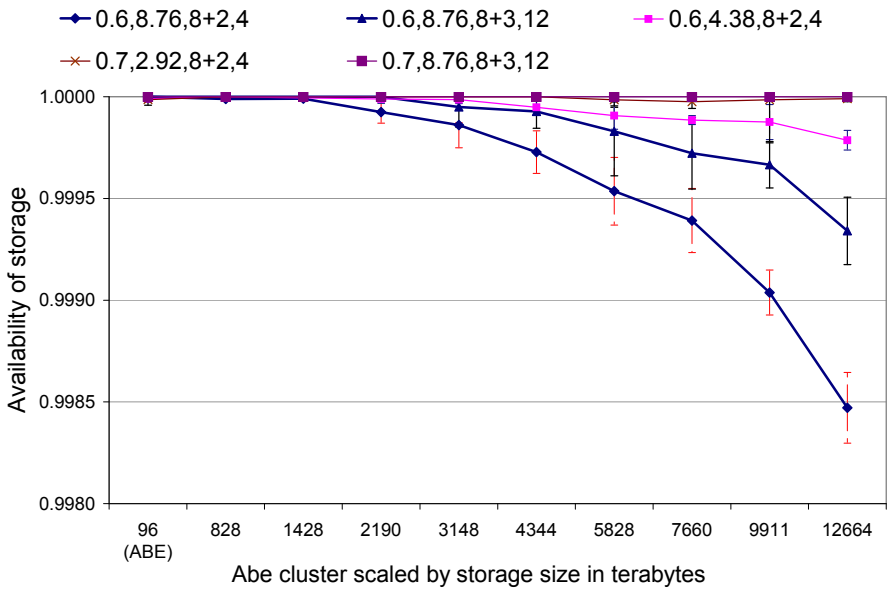


Figure 2.2: Availability of storage with respect to disk failures; Label with values (0.7,2.92,8+2,4) represents a tuple =(Weibull shape parameter β, AFR in %, RAID configuration, average disk replacement time in hours)

hardware as one scales the file system from the current 96TB (Abe's file system) to 12PB (the Blue-Waters file system). The key observation is that the RAID6 architecture provides sufficient redundancy and recovery mechanisms to mitigate the impact of high disk failure rates to a very large extent. First, note that all configurations of failure and recovery rates for an Abe-sized cluster file system have nearly 100% availability (refer to the first data point in Figure 2.2). However, as the experiments are scaled from Abe's system to a peta-scale system, our simulation results show that the RAID6 architecture cannot provide the same level of storage availability for some of the failure model configurations. The SAN architect's plan to use (8+3) RAID in Blue Waters is important; it provides better reliability than the (8+2) RAID on peta-scale systems. While RAID6 provides a larger margin for disk failure rates, i.e., up to 8.6% AFR, it is very important that these rates be contained to lower thresholds by disk manufacturers and vendors to provide the adequate level of availability. If one makes a pessimistic assumption of a higher infant mortality rate in disks (Weibull shape parameter = 0.6), the availability falls below 99.9% for peta-scale storage.

To better understand the cost of disk replacement, we compute the expected number of disks that need to be replaced per week for the RAID6 tiers. Figure 2.3 depicts the

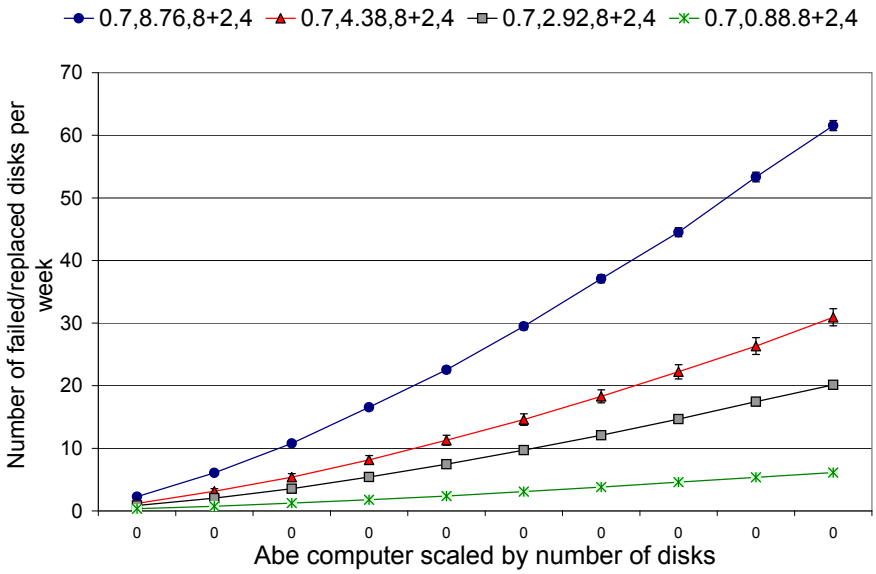


Figure 2.3: Average number of disks that need to be replaced per week to sustain availability

average number of disks that need to be replaced per week to sustain the availability so that the CFS does not suffer failure due to RAID6 failure. The configuration (0.7,2.92,8+2,4) corresponds to the Abe cluster with 0 to 2 disk replacements per week. Each time a disk fails, there is an operational cost (in dollars) that is borne by the SAN vendors as they provide extended support to their SANs. As the CFS system is scaled to support peta-scale computers, the number of disks that need to be replaced increases, increasing the labor cost and the replacement cost. Therefore, the SAN vendors have an incentive to increase the disk MTBF to reduce their overall support cost.

Survival analysis of the disk failure data provided a good estimate of the Weibull distribution's shape parameter β, but the estimate for the scale parameter (MTBF) was insignificant [37]. Using simulations, we estimated an MTBF that matched the average disk failures per week for the scratch partition and determined that an MTFF=300,000 hours or an annualized failure rate (AFR) = 2.92% to be a good fit.

2.6.2 CFS Availability and CU

To analyze the impact of all components that determine the availability of the CFS, we evaluated the availability and CU of the Abe system. The experiments are scaled to the size of a peta-scale computer to allow understanding of the impact of failures on those measures. Figure 2.4 shows that the CFS availability decreases as one scales the system to support a peta-scale computer. Since most of the parameter values were obtained through the log data analysis and times reported by SAN administrators, our measures for CFS availability matched with Abe's availability as shown in Table 2.2. Therefore, we have higher confidence in the measures of availability and CU as we scaled the models to represent a peta-scale computer with a petabyte storage system. The storage availability in Figure 2.4 refers to configuration (0.7,2.92,8+2,4), which models the Abe cluster's current environment. We find that the RAID6 subsystem in this configuration continues to provide an availability of 1, but the CFS availability is reduced from 0.972 to 0.909 as one scales the design to support the peta-scale system. The reduction is mainly due to correlated failures in OSS and hardware. Improving upon Abe cluster's design, the architect could provide an additional standby or spare OSS to replace the failed OSS quickly. Our evaluation shows that this approach can improve the availability by 3%. To improve the availability further, the architects have to develop solutions to mitigate correlated errors. For example, improving the robustness of the Lustre-FS can reduce the software-correlated errors. The CU in Figure 2.4 shows that the cluster's network architecture between the compute nodes and the CFS has a profound impact on the cluster utility available to the users. The trend to move away from customized backplanes to COTS network hardware (with its complicated software stacks) has decreased the CU. The transient errors seen in the network can be mitigated by providing multiple network paths between the compute nodes and the CFS.

2.7 Conclusion

Many researchers have focused on developing and understanding dependability of clusters for supercomputing applications. In this chapter, we have taken steps to understand the

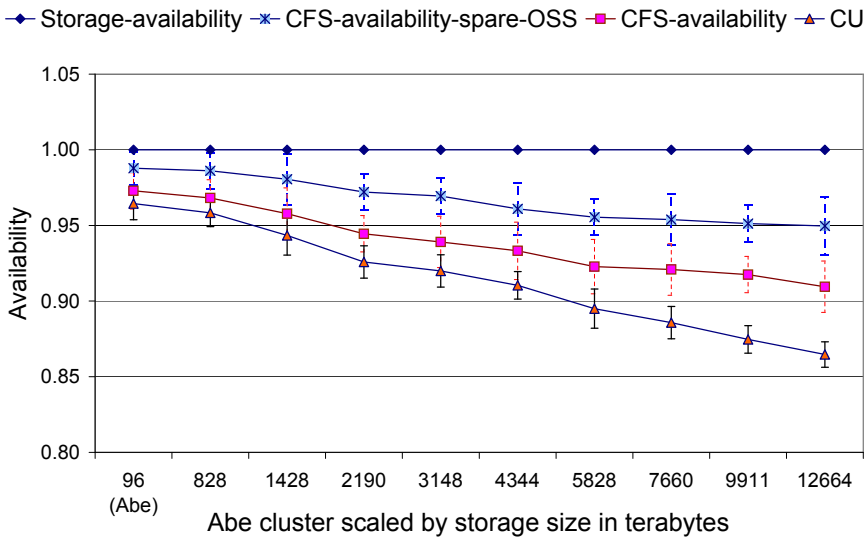


Figure 2.4: Availability and utility of the Abe cluster when scaled to petaflop-petabyte system

dependability and availability of the ABE cluster through failure data analysis and discussions with administrators at multiple levels of the cluster operation, from the lowest level of the SAN's availability, to the cluster's availability, and to user perception of cluster utility at the top level. Our key findings through analysis and simulation showed that the RAID6 design for a disk's dependability has limited the impact of disk failures on the CFS, even when the model is scaled to evaluate the support for peta-scale systems. On the other hand, transient errors, hardware errors, and software errors contribute significantly to failures, and these components are the limiting factors for the high availability of the CFS. We believe that peta-scale architects will have to focus on those issues to develop solutions to improve the overall availability of the CFS.

More generally we have shown the success of component-based system modeling, combined with data analysis for real systems, for evaluating the availability of the ABE's CFS. We will extend this method in later chapters to model more complex storage systems.

CHAPTER 3

MODELING STORAGE SYSTEM FAULTS

3.1 Introduction

More so than in the past, the high-performance computing systems of tomorrow will grow in computing power and storage space far faster than the growth of per-chip computing power, and storage density. Today's large scale systems are dominated by components assembled from commodity PC hardware, aggregated into large clustered systems [11, 39]. Node counts for state of the art terascale systems number in the tens of thousands, and peta-scale systems are likely to contain hundreds of thousands of nodes. Under these conditions the normally rare mean time to failure (MTTF) of various components compose to create a short MTTF for the first failure in the composed system. Faults that previously were unlikely to occur within the mean life time of a system also scale up in frequency system wide, presenting new challenges which current fault tolerance methods do not address [11, 40]. Understanding these failure models is a key first step towards developing tools to evaluate their impact, and the effectiveness of proposed methods for coping with these failures.

In Chapter 2 we explored a component based model of a large-scale storage system, using only whole disk failures. In this Chapter we expand our portfolio of fault-models to include latent sector errors (LSEs) and undetected disk errors (UDEs). Our fault model for LSEs is derived from the existing fault models in the literature, while our the UDE fault model we present is a novel model which we will derive from various data in the literature and an understanding of the physical manifestations of UDEs. These fault models will be used along with whole disk failures in subsequent chapters to model storage system reliability in more complex fault environments.

3.1.1 Challenges

Understanding how faults change with larger scale systems faces many challenges, chief among them being the unwillingness of system owners to publicly report failure data [41]. Vendor specified parameters are thus the most common source for failure models. These parameters, most commonly given as the MTTF, are of dubious accuracy, often based on back of the envelope calculations, giving radically different values, depending on the assumptions made. This has led to a high level of skepticism towards these vendor derived figures [42, 7]. Coupled with this is the large discrepancies between vendor reported MTTF and user reported MTTF [43, 44].

3.1.2 Profiling Failures in Large-Scale Systems

It has been shown by [41] that disks are the most commonly replaced component of large scale systems. This is in part due to their high population within a system, compared to other hardware components. Other commonly replaced components include system memory, power supplies, and motherboards or CPUs. Given the current lack of detailed failure data for components other than those of the storage system in large-scale systems, we will focus our study on storage level failures.

While in the past storage failures were viewed as an all-or-nothing scenario, i.e. either the drive was working perfectly, or not at all, these models have proven inaccurate for large-scale systems. Modern storage systems are known to suffer from more complex faults, which primarily fall into one of two categories: LSEs, and UDEs [45]. In the following sections we will investigate each of these failure models.

3.2 Latent Sector Errors

3.2.1 Introduction

While LSEs do not typically contribute to the MTTF of the entire drive, they do influence the reliability of data on a given drive, creating a new metric for failure on a drive, the mean

time to data loss (MTTDL). While many modern storage systems rely on RAID to provide redundancy, faults such as LSEs are somewhat orthogonal to those issues solved by RAID, especially since a latent sector error is not evident until the affected sector is accessed after the fault occurs [9]. LSEs are orthogonal to RAID given the fact that a single latent sector error has the potential to lead to data loss during a RAID group reconstruction [46]. LSEs are distinguished from other types of disk faults based on the way a disk responds in the presence of the fault. LSEs specifically refer to those cases where a sector cannot be read or written, or when the sector reports an uncorrectable ECC error [9]. In all of these cases, the sector is considered lost.

LSEs seem, at least in part, to be affected by the manufacturing quality of the disk in question. While 8.5% of near-line disks were found by [9] to suffer from a latent sector error, only 1.9% of enterprise class disks exhibited this behavior. Both classes of disks are more likely to develop LSEs with time, but the proportion of near-line disks which have suffered a latent sector error was shown to grow more rapidly than enterprise class disks. Disk size was found to have an effect as well with the proportion of disks suffering a latent sector error increasing with capacity, however the proportion of LSEs per Gigabyte does not change appreciably with increased capacity [9]. While the quality of the drive matters with respect to the first latent sector error [9] found that for disks with at least one error, near-line drives were equally likely to have additional LSEs when compared to enterprise class drives.

3.2.2 LSE Model

LSEs have been shown to exhibit spacial locality [9]. According to [9], given a single latent sector error, the probability of finding one or more LSEs within a 10MB radius of the existing error is 0.5. LSEs were also found to exhibit temporal locality, [9] found that between 40% and 80% of errors occurred within one minute of a previous error. Interarrival times were classified using two measures, the temporal locality exhibited by arrivals, and a measure [9] calls *decay*, which is a measure of the time taken to develop n additional LSEs, as measured from the first latent sector error on the disk. In addition to a high level of temporal locality, it was found that 54.8% of near-line, and 62.0% of enterprise class disks developed at least

one additional error within one month. 5% of near-line and 10% of near-line disks developed at least 50 additional errors within one month. Interestingly, the age of the disk does not seem to have a direct effect on *decay*. Most additional faults will develop within the first month after the initial fault.

3.2.3 Mitigation

RAID has been shown to be insufficient for prevent the loss of data from LSEs [46], and even a full scrub detected only 86.6% of LSEs on near-line disks, and 61.5% of LSEs on enterprise class disks. Read operations were likewise found to detect 13.4% and 19.1% of LSEs for near-line and enterprise drives. Writes do not detect latent sector errors for near-line disks, due to near-line disks automatically reassigning bad sectors, but in enterprise disks detected 19.3% of LSEs. Overall standard mitigation methods found 77.4% of LSEs during normal operation.

3.3 Undetected Disk Errors

3.3.1 Introduction

Despite the reliability of modern disks, recent studies have made it increasingly clear that a new class of faults, that of *UDEs* are a real challenge facing storage systems as capacity scales [13, 47, 40]. While RAID systems have proven quite effective in protecting data from traditional failure modes [12], these new silent data corruption events are a significant problem unaddressed by RAID [13]. UDE faults themselves are drawn from two distinct classes, *undetected read errors* (UREs) and *undetected write errors* (UWEs). UREs manifest as transient errors, and are unlikely to effect system state beyond their occurrence. UWEs, on the other hand, are persistent errors which are only detectable during a read operation subsequent to the faulty write, and thus manifest in a similar manner to a URE [40]. Metrics to quantify the occurrence of data corruption due to UDEs have not been presented before since these events have been deemed to be very rare. Recently capacity scaling has

made UDEs common enough to be a concern and drives further study both on the rate of occurrence of these faults, and their manifestation as errors from a user perspective.

While both evidence of UDEs and suggested techniques for their mitigation have been outlined in the literature [13, 40], it is clear that there are several situations where many of these techniques will not be able to prevent UDE induced faults from manifesting as errors [47].

In order to fully understand the effect of UDEs and how they manifest in a given storage system, one must model the disks at a fine level of detail, looking at individual block read and write information in order to track portions of the disk which are suffering from a UDE, the propagation of UWEs due to normal operations within the RAID system (including rebuild events), and the effectiveness of mitigation techniques to prevent a UDE based fault from manifesting as an error.

3.3.2 UDE Model

A growing concern in the storage community has been errors for which RAID [12, 45] does not provide adequate protection. Schroeder and Gibson note that despite a supposed MTTF for drives ranging from 1,000,000 to 1,500,000 hours field data suggest that MTTF is, in practice, much lower [41]. By analyzing data corruption events for over 1.53 million disks in production storage systems Bairavasundaram et al. documented cases of data corruption events that occurred in a 41 month period. This work illustrates the existence of several types of rare faults which manifest as corrupt data blocks. These errors were detected by their production system due to additional detection mechanisms implemented at the file system layer by their system. We refer to these silent data corruption events [48, 45] as UDEs [40].

UDEs can be divided into two primary categories, UWEs and UREs [40], and arise from a variety of factors, including software, hardware and firmware malfunctions. The primary difference between UWEs and UREs is that UREs are transient in nature, while UWEs result in a changed system state that can cause many subsequent reads to return corrupt data. Table 3.1 summarizes the primary types of UDEs, and how they manifest as actual

Table 3.1: Summary of UDE types and manifestations

I/O Type	*UDE type*	*Manifestation*
Write (UWE)	Dropped Write	Stale data
	Near off-track write	Possible stale data on read
	Far off-track write	Stale data on intended track
		Corrupt data on written track
Read (URE)	Near off-track read	Possible stale data
	Far off-track read	Corrupt data

errors on the disk [40].

Dropped writes occur when the write head fails to overwrite the data already present on a track. In this case the disk remains in its previous state as if the write never occurred [40].

Off-track writes (also referred to as misdirected writes) come in two categories, near and far off-track. Both types occur when the write head is not properly aligned with the track. In the case of a near off-track write, the data is written in the gap between tracks, adjacent to the intended track. On a read operation, the head may align itself to either the target track, or the off-track, potentially producing stale data. Far off-track writes occur when data is written even further off-track, such that it corrupts data in another track entirely. Subsequent reads to the track which was mistakenly written will produce corrupt data, while reads to the track which was intended to be written will produce stale data. [40]

It is important to note, however, that not all UWEs introduced disk errors manifest as a user level undetected data corruption error. A subsequent (good) write to the affected track will remove the error, preventing undetected data corruption. Likewise a near off-track write will cause future reads to randomly return either good or stale data. Though a far off-track write results in both stale data on the intended track, and corrupt data on the unintended track, if the unintended track is not currently in use, that part of the effect will be mitigated. Likewise, a UWE may manifest as several undetected data corruption events if the same track is read multiple times before it is corrected by a subsequent write operation. Thus a one-to-one correspondence does not exist between UDEs and user level undetected data corruption errors.

3.3.3 UDEs in RAID Storage Systems

Systems designed for high reliability often make use of RAID storage systems. While there has been little study of the effect of UDEs in RAID systems, it has been shown that data scrubbing, the normal disk error mitigation and detection technique in such systems, is not sufficient to protect against all UDEs [47, 40]. Even under RAID6, the most powerful RAID technique in common usage, a data scrub may incorrectly assess the integrity of data due to a UDE [40], and in some cases even propagate the error.

3.3.4 Modeling UDEs

Storage systems faults are complex events, arising from a combination of hardware, software, and firmware malfunctions. In the case of well studied faults, such as latent sector errors, and complex models have been derived to capture detailed relationships and correlations. Bairavasundaram et al., when designing a model for latent sector errors, capture temporal and spatial locality measures [9]. Schroeder and Gibson in their study of MTTF propose and fit both Weibull and Gamma distributions, obtaining better results than with exponentially distributed inter-failure times [41]. Unfortunately when it comes to UDEs, there is very little information on their rates of occurrence, let alone enough data to fit multiparameter distributions, or ascertain spatial and temporal relationships on their failure. UDEs could be modeled with exponential inter-failure times, making no assumptions with respect to temporal and spatial locality.

The rates of UDE occurrence were estimated by using a combination of the data presented in [9] and the data presented in [13], and are summarized in Table 3.2.

Table 3.2: Estimated rates of UDEs in $\frac{\text{UDEs}}{\text{I/O}}$.

	Estimated Rate	
UDE Type	**Near-line**	**Enterprise**
Dropped I/O	$9 \cdot 10^{-13}$	$9 \cdot 10^{-14}$
Near-off Track I/O	10^{-13}	10^{-14}
Far-off Track I/O	10^{-12}	10^{-13}

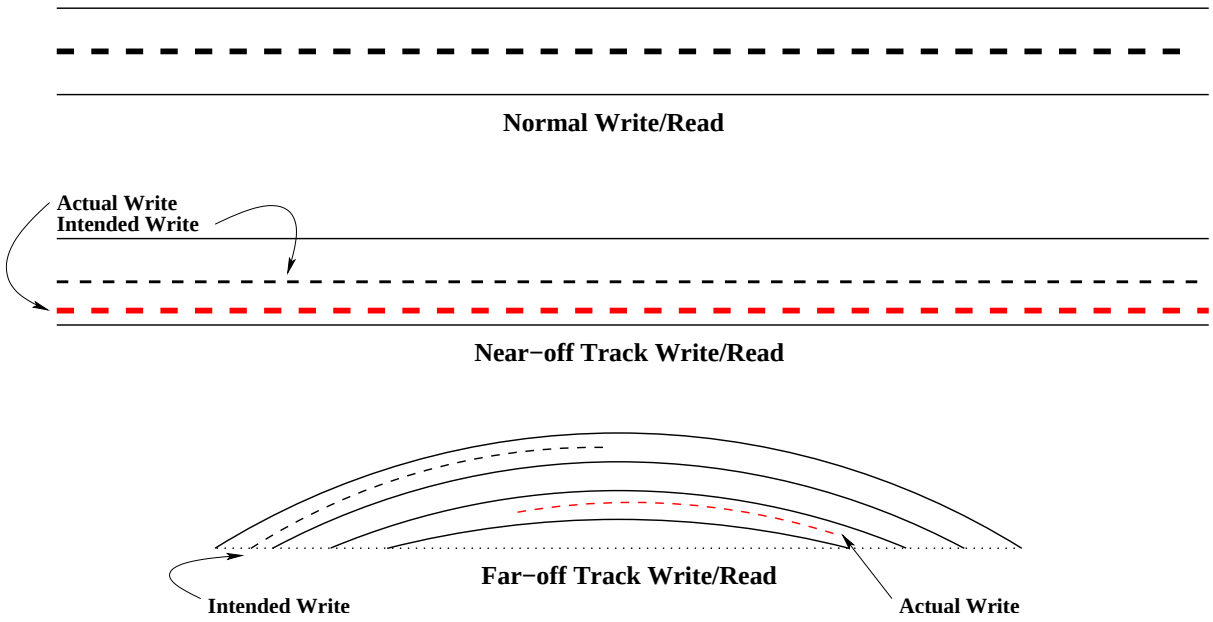


Figure 3.1: Disk tracks after a normal I/O operation, near-off track fault, and far-off track fault.

3.3.5 Mitigation

Despite the threats to data integrity and storage system reliability posed by UDEs, production systems rarely implement detection and mitigation strategies [49]. Methods have, however, been proposed, if not evaluated in real systems. One family of proposed methods to mitigate the effect of UDEs in RAID are of the form of *parity appendix methods* [40]. In these methods, meta-data is co-located with some or all of the blocks associated with a chunk of data and its parity blocks. In our models we focus on a data parity appendix method [50, 40], which utilizes the meta-data portion of a block to store a sequence number. This sequence number is the same for all blocks in the write. UDEs can be detected with this method by comparing the sequence numbers stored in the parity and data blocks. The sequence numbers won't match when a UDE manifests as an error unless a collision occurs in the sequence number with probability

$$P(\text{Seq(Parity)} = \text{Seq(UDE)}) = \frac{1}{2^b}$$

where b is the number of bits allocated for the sequence number. On a read, the sequence number for each data block is retrieved from parity and compared, allowing this technique to mitigate UDE manifestations, at the cost of an extra read.

3.3.6 RAID and Parity Pollution

The more typical method for ensuring data integrity and reliability is to employ some form of parity scrub. In these cases a scheduled process checks stored data against the relevant parity strips in a RAID subsystem. While this can detect UDEs under normal operations, it cannot reliably detect all data corruption due to UDEs. The scrub can mistakenly find data or parity to be correct when both are wrong. Even in cases when data corruption from a UDE is detected, it often cannot be corrected. To understand these properties of UDEs we provide one example each for a RAID5 and RAID6 system. We consider a RAID stripe to be the minimum set of data blocks and parity blocks from a set of disks which are related in the reliability framework. Read operations are processed by reading the relevant data from the relevant disks. Write operations cause updates to both data and parity blocks in the form of a *read-modify-write* operation.

Given a RAID5 system, and a RAID strip consisting of data blocks, A, B, and C and parity block P we represent the data values in A, B, and C as a, b, and c and use the symbol $\oplus$ to represent a XOR operation. Figure 3.2 shows an example series of operations that leads to an inconsistent parity state known as *parity pollution* [40]. The stripe begins in the state represented by the tuple

$$(a_0, b_0, c_0, a_0 \oplus b_0 \oplus c_0)$$

.

The second row in the figure shows a write operation, intended to update the block A to a_1, which suffers a UDE in the form of a dropped write. This failure is not detected by the system. The parity disk is read, and modified as follows

$$(a_1 \oplus a_0) \oplus (a_0 \oplus b_0 \oplus c_0) \rightarrow a_1 \oplus b_0 \oplus c_0$$

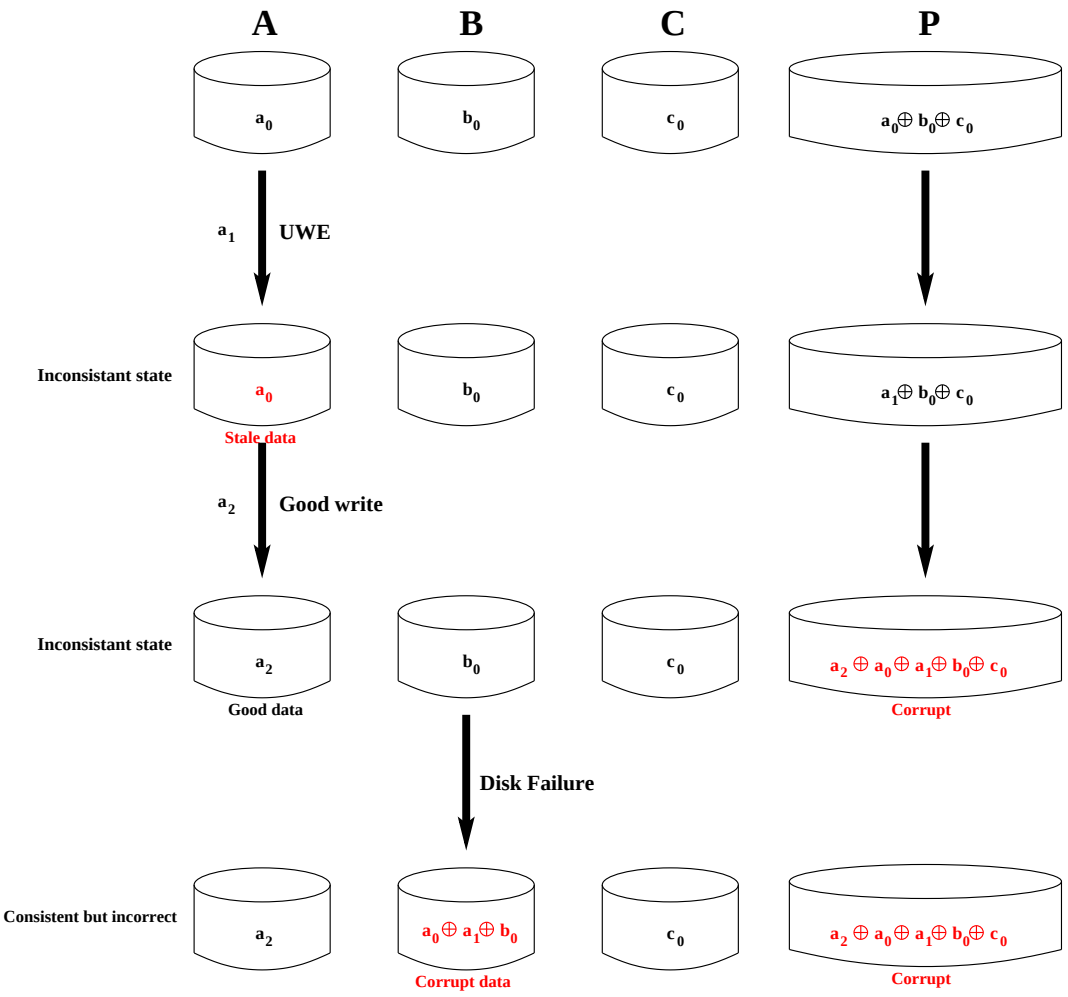


Figure 3.2: Example parity pollution due to a dropped write. The first step has a UWE during a write; the the second step is a normal write; the third step represents a disk failure; in the final step the disk is rebuilt incorrectly.

. The parity is in a state which would be correct if a_1 had been properly written, but is inconsistent with the state of the stripe at present. A scrub at this point would detect the inconsistency, but not where the error had occurred. A disk scrub would result in marking the entire stripe as bad resulting in data loss for the entire stripe. The same situation would occur if the data block was written successfully but the read-modify-write had resulted in a dropped write to the parity block. The next row of the figure represents a successful write which updates the strip A to a_2. The read-modify-write operation to the parity disk, however, produces an interesting result. The data block A is read as a_0 before the write, resulting in the following modification of the parity block:

$$(a_2 \oplus a_0) \oplus (a_1 \oplus b_0 \oplus c_0) \rightarrow a_2 \oplus a_0 \oplus a_1 \oplus b_0 \oplus c_0$$

which is then written back to P. Though the entire operation was error free it has resulted in pollution of the parity block which is now in a state which is inconsistent with the remainder of the disk. The fourth row shows the state of the RAID stripe after the disk on which B resides has failed, and a subsequent rebuilt operation has been performed. Even though the stripe is now consistent, the B block contains an error as it has been rebuilt as $b_0 \oplus a_0 \oplus a_1$ instead of b_0. This error is now undetectable by a parity scrub.

This shows how a single UWE can combine with otherwise normal operations on a RAID5 system to result in a state in which an error is present, but undetectable by any normal means including a parity scrub. A similar situation can occur for RAID6 systems, and is detailed in [40].

3.4 Conclusions

In this section we have detailed two new important, but rare, faults, LSEs and UDEs and the ways in which they can occur on disks. We have reviewed the fault models for LSEs presented in the literature and detailed a novel fault model for UDEs based on existing data and an understanding of the ways in which UDEs manifest. We will use our UDE fault model in Chapter 4, and both the LSE and UDE fault models in Chapter 6 to develop an

understanding of how these faults, combined with whole disk failure, impact the reliability of large scale storage systems.

CHAPTER 4

SIMULATING THE EFFECTS OF UDES ON LARGE-SCALE STORAGE SYSTEMS

4.1 Introduction

In Chapter 2, we discussed building component-based models of large-scale systems using real data. In Chapter 3 we discussed a new class of faults, UDEs. In this chapter we combine these methods to study the reliability of a set of storage system designs in the presence of UDEs to understand their impact under various real world workloads, modeled at a component-based level.

Recent studies have made it increasingly clear that UDEs are a real challenge facing storage systems as capacity scales [13, 47, 40]. While RAID systems have proven quite effective in protecting data from traditional failure modes [12], these new silent data corruption events are a significant problem unaddressed by RAID [13]. Metrics to quantify the occurrence of data corruption due to UDEs have not been presented before since these events have been deemed to be very rare. Recently capacity scaling has made UDEs common enough to be a concern and drives further study both on the rate of occurrence of these faults, and their manifestation as errors from a user perspective.

While both evidence of UDEs and suggested techniques for their mitigation have been outlined in the literature [13, 40], it is clear that there are several situations where many of these techniques will not be able to prevent UDE induced faults from manifesting as errors [47]. Given that the rate of UDE occurrence is low, testing these techniques in a real system would be costly, likely requiring a prohibitively large array of disks to witness UDEs in large enough quantities to derive measures of their effects within a reasonable period of time. A need therefor exists for modeling tools which address the particular challenges presented by such systems. A model provides the capability to analyze the risk posed by UDEs in a

given system and the fault tolerance coverage provided by suggested mitigation techniques. A simple analytical model could fail to account for important emergent trends in UDE manifestation for a given workload. For example a block that has had a UWE error must be read before that fault manifests as data corruption and therefore the chances of system failure depend on the I/O stream in question. A simulation approach that is customizable and can determine the effect of UDEs for an arbitrary workload is an effective way to capture this behavior.

Efficient simulation of UDE faults in large scale systems proves to be a serious challenge, however, due to the *stiffness* of the system. A system is said to be *stiff* when events occurring on vastly different timescales must be simulated in order to capture the behavior one wishes to model. In order to fully understand the effect of UDEs and how they manifest in a given storage system, one must model the disks at a fine level of detail, looking at individual block read and write information in order to track portions of the disk which are suffering from a UDE, the propagation of UWEs due to normal operations within the RAID system (including rebuild events), and the effectiveness of mitigation techniques to prevent a UDE based fault from manifesting as an error. This means any model of such processes will necessarily consider events on two very different scales, from the very fast block level I/Os (with a rate of roughly 100 ios/sec), to the much rarer UDEs (with a rate of roughly 10^{-12} UDEs/io).

In order to satisfy these requirements we present in this chapter an extensible hybrid framework for simulating large scale storage systems that combines discrete event simulation on multiple levels of resolution with numerical analysis in a hybridized fashion. Our methods take advantage of dependency relationships within a the I/O stream to temporally switch between numerical methods, a block level discrete event simulator, and a hybridized numerical model with some discrete events simulated. Our techniques achieve a more efficient use of time and space than discrete event simulation alone. These methods have been implemented in a simulator which takes as input a model of a storage system and a model of a workload, and produces as output an estimate of the rate at which UDEs manifest as corruption at the user level. Our simulator has been designed in a way that allows the user to build component level models of storage systems, extend components to create new

behavior, and test under arbitrary workloads and rates of UDEs. Mitigation techniques can be easily implemented by extending the implemented base classes for disks.

The chapter is organized as follows: We will then a model for UDEs for use in our simulation, this derivation includes determining appropriate rates for the various types of UDEs, as these rates have yet to be determined in the literature. Next we will discuss how we create workload and system models models for a novel hybrid simulator and the operation of the simulator itself. Finally, we present our results which illustrate the problems posed by UDEs as systems scale, and the effectiveness of even simple mitigation techniques in preventing the faults injected by UDEs from manifesting as undetected data corruption events.

4.1.1 Modeling Considerations

Given our assumptions that UDE manifestation is likely highly correlated to the workload and system in which they occur, we take a flexible approach in building our system models, and solution techniques. Our solution techniques do not make any assumptions about the underlying workload or model, allowing the user to construct appropriate workloads or models based on their own needs or assumptions. We provide a common interface for workload models, allowing the use of traces, or probabilistic models (as was used in our evaluation), and a framework for constructing a storage system model by composing common component level objects as detailed in Chapter 2, which can be subclassed as needed to increase the library of components from which the system can be built.

Once a workload model and system model have been created, we utilized a multi-modal simulator which executes the model at three different resolutions, based on a dependency relationship with UDEs which occur in the I/O stream.

4.2 Workload Modeling

The manifestation of UDEs as actual data corruption events is dependent on read operations following a write which suffers from a UDE (or in the case of UREs, just simply a read operation). This fact makes modeling the workload for the system an important consideration.

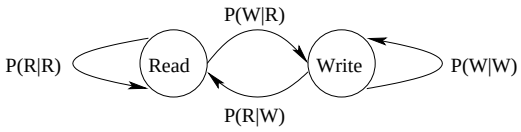


Figure 4.1: Two state DTMC read/write workload model.

The mix of reads and writes impacts the manifestation of UDEs, as does the correlation of reads and writes in time. While this fact drives a desire to use realistic representative work loads in any simulation, perhaps even traces, the rare nature of UDEs makes makes this a difficult proposition. Utilizing a trace based simulation has two problems. The first is a lack of sufficiently detailed traces, and the second is a lack of traces of sufficient length. Due to the long periods of time involved in simulating UDE mitigation, one requires hundreds, if not thousands, of days of block level traces. Additionally, trace based analysis is limited to a handful of possible trajectories in the file system, only those captured in the available traces. Given these limitations we form a simple probabilistic model of the workload which we use to generate synthetic workloads that statistically match available traces.

Generating a representative synthetic workload has proven difficult in the past, and is still largely considered an open problem. For the purposes of this analysis we present a simplified workload model which captures the primary metrics of interest for our needs, namely the ordering of reads and writes, which can be directly parametrized using real traces to which we have direct access. Despite our selection of this method of workload modeling, our simulator has been designed in a flexible manner which allows it to utilize any workload model a user would like to design. The main simulator couples with the workload model simply by requesting events from the workload model, including the type of I/O, and size and location of the write, and the time until the next I/O. It then injects UDEs as appropriate.

In order to characterize the workload used in our study, we create a statistical model of five primary features from a given trace.

- **Per Disk Rate of I/O -** Given that our UDE rates are in units of $\frac{UDEs}{I/O}$, it is important to know the rate at which I/O is conducted in our workload. This rate is captured in the workload parameter io/s. Our workload model assumes exponentially distributed inter-arrival times, with rate io/s.

- **Diversity of the I/O Stream -** Just knowing the rate of incoming I/O requests is not, however, enough to characterize the system. We also need some measure of the diversity of the I/O stream. While io/s measures the number of I/O requests per second, it tells us nothing about the average number of unique chunks on which I/O operations are performed, per second. We introduce the parameter uc/s to model this, which is a measure of the unique chunks read or written each second.

- **I/O Size -** Based on our analysis of some available traces we model the size of an I/O operation within the simulator as a Exponential random variable with a mean request size of $size_{\text{i/o}}$ blocks.

- **Read/Write Composition of the I/O Stream -** When the simulator is not suffering from a UDE, our simulator discards I/O events rather than simulate them, as they cannot have an effect on the rate of undetected data corruption errors. When an UDE does occur, it is important to know whether it occurred during a read operation (and is thus transient) or during a write operation, (and could thus lead to a series of data corruption events). Towards this end we estimate the parameter $P(R)$, which tells us the probability with which a given I/O is a read request.

- **Correlation of I/O Stream Composition -** In our analysis of available traces, we found that an I/O request for an individual chunk is highly temporally correlated with the most recent I/O for that same chunk in the trace. In order to model this behavior we utilize a discrete time Markov chain (DTMC) for each active chunk to determine the next I/O operation that will take place after a read or a write, as illustrated in Figure 4.1. This model has four transition probabilities which we estimate from the trace data. $P(R|R)$ is the probability of a chunk being read if the most recent I/O for the chunk was also a read. $P(R|W)$ is the probability of a chunk being read when the most recent I/O request was a write. $P(W|R)$ is the probability of a chunk being written when the most recent I/O request was a read, and finally $P(W|W)$ is the probability of a chunk being written when the most recent I/O request was a write.

Table 4.1 indicates the values of these parameters for three different workloads. The first

Table 4.1: Parameters for the workload models.

Parameters	**Abstract**	**Write Heavy**	**Read Heavy**
io/s	100	122.04	90.24
uc/s	10	16.2623	7.2226
$size_{io}$	4096 B	3465.98 B	2448.94 B
$P(R)$	0.6	0.23161	0.82345
$P(R\|R)$	0.6	0.4488	0.829483
$P(R\|W)$	0.6	0.8339	0.204677
$P(W\|R)$	0.4	0.5512	0.170517
$P(W\|W)$	0.4	0.1661	0.795323

workload, which we call the Abstract Workload, is derived from parameters assumed in the calculations presented in Section 3.3.4 derived from [13] and [41]. Both the "Read Heavy" and "Write Heavy" workloads are estimated empirically from actual disk traces. Separate DTMCs are kept in the workload model for each chunk on the disk that is being read or written.

4.3 Disk Model

In order to understand the manifestation of UDEs into user-level undetected data corruption errors, we present a block level model of storage systems which can be used to understand the actual effects of these faults in real systems. This block level model is designed to be easy to extend and compose with other similar models to allow for the construction of more complex disk models. Each individual block level model keeps track of the state of a single disk. Models can be composed to allow the simulation of various RAID configurations and techniques for detecting UDEs. In our models, for example, a RAID6 system is simply a subclass of a disk, which contains a number of composed disk models. RAID6 with mitigation is simply a subclass of our RAID6 model which implements the sequence number data parity appendix method described in Section 3.3.5.

4.3.1 Block-Level Model

We model each block on the disk as a non-deterministic finite automaton (NFA). The set of states Q of the NFA represents the states of the block, and is comprised of the union between a set of stable states Q_{stable} and unstable states Q_{unstable} defined by

$$\begin{aligned} Q &= Q_{\texttt{stable}} \cup Q_{\texttt{unstable}} \\ Q_{\texttt{stable}} &= \{\texttt{good}, \texttt{stale}, \texttt{corrupt}\} \\ Q_{\texttt{unstable}} &= \{\texttt{stale-good}, \texttt{corrupt-good}, \\ &\quad \texttt{corrupt-stale}\} \end{aligned}$$

The state of the block is in the state `good` only if no fault has occurred for that block. The state `stale` represents a faulty block state after either a dropped write, or a far-off track write which was intended for the block in question. The state `corrupt` represents a faulty block state after a far-off track write which was intended for another block, but was written to the block in question instead.

Unstable states differ from stable states in that the behavior of a read operation is ill-defined and may return either good data, or corrupt/stale data. They are used to model the effects of near off-track writes. In the case of an unstable state where one of the tracks is `good`, a read to that track will not necessarily manifest as an error. We make the assumption that reading either track is equally probable.

The unstable state `corrupt-corrupt` is omitted from this set, as it is equivalent to a `corrupt` state, as is `stale-stale` for similar reasons.

The sector model transitions from state to state on a set of input symbols Σ, which correspond to the write events to the sector, both faulty and non-faulty. The set of faulty write events corresponds to the set of UDE events discussed in Section 3.3.2, which is shown below.

$$\Sigma = \{\texttt{GW}, \texttt{DW}, \texttt{N-OTW}, \texttt{F-OTW}_H, \texttt{F-OTW}_O\}$$

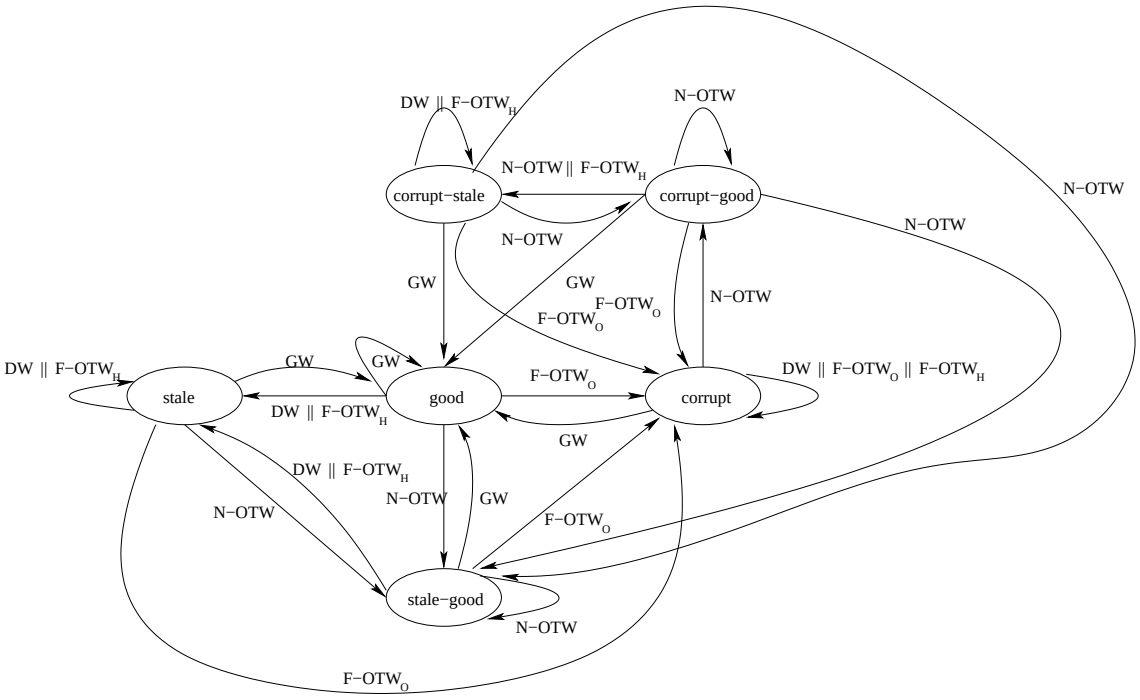


Figure 4.2: NFA representation of transitions in the block model.

Symbol `GW` represents a good write event. `DW` represents a dropped write event. `N-OTW` represents a near offtrack write event. `F-OTW`$_H$ indicates a far offtrack write event which was intended to write to this block but instead wrote to another. `F-OTW`$_O$ represents a far offtrack write event which was intended for another block but was instead incorrectly written to the block in question.

Figure 4.2 shows the NFA states with transitions for a single block, indicating how the block changes states due to UDE events. Of particular importance is the fact that from all states a good write event brings us to the `good` state, masking the UDE. This fact is important when considering how a UDE fault manifests as an actual error. In the case that a UDE occurred, but the same blocks were again written with no error before a read for those blocks was issued, the fault will not manifest as an error.

4.3.2 Disk-Level Model

In order to represent an entire disk, the block level model indicated in Section 4.3.1 is implemented for every block in a disk, forming our base disk model. Each block is indexed,

and thus can have individual write operations directed to it, causing state changes in the underlying automata. For efficient storage of the automata, we take advantage of the fact that, barring a UDE which is by definition a rare event, the entire disk consists of blocks in the `good` state. By representing the state of all blocks in a disk using a sparse matrix with `good` as the default value, we achieve a space efficient implementation of even very large disks. This sparse matrix is represented with a hash-table, using a hash function of $LBA \bmod n$, for some n. While a simple function, it exploits the fact that UDEs will most often manifest on a disk as a string of consecutive blocks in a state other than `good`. The modulo function then works to try to ensure that consecutive blocks will appear in different elements of the hash-table, which keeps access times for the hash table at a minimum. This improves the time it takes to update the state of our various block level NFAs, and thus overall simulation time.

Individual disk models are composed with a controller to form RAID units. The controller model presents to the simulator an interface to write to the underlying disks as if they were a single larger disk. It performs the calculations of which stripe to write to on a block by block basis to enforce the RAID model. The RAID interface also serves to inform the simulator of the number of actual disk reads and writes executed per requested operation so that UDEs occurring due to a single logical write to a RAID system will be generated appropriately for the larger numbers of actual disk reads and writes performed due to RAID operations such as read-modify-write. Each RAID unit undergoes weekly scrubbing, and we make the assumption that during a scrub all UDEs will be detected and marked unreadable so as not to result in a data corruption event in the future.

Parity scrub mismatches are handled by marking the affected blocks as unreadable, meaning that any reads from the workload stream to these blocks will not be executed. On a successful write operation (UDE or otherwise) to a block, this status is changed via the NFA in Figure 4.2 treating the unreadable sector as being in the good state just before the write.

4.4 Simulation Framework

Given the rare nature of UDEs, attempting to calculate the mean rates of undetected data corruption due to their injection into a storage system using just discrete event simulation would prove prohibitively costly. With the rates given in Table 3.2, it is not unreasonable to expect 10^{13} or more I/O events in between UDEs. Even given a discrete event simulator capable of processing 10^7 or more events per second evaluating a single UDE event would take almost a month on modern workstations. This makes simulating a large number of UDEs impractical, and seems to preclude their analysis via simulation.

We solve some of the efficiency issues inherent in simulating a stiff system by designing our simulator to have three operating modes (illustrated in Figure 4.3), and allowing the simulator to dynamically switch between modes to maximize efficiency while maintaining accuracy in estimating the rate of undetected data corruption served to the user. While our simulator improves our simulation time, it makes the assumption that the underlying storage system consists only of basic RAID subsystems and that each subsystem is independent.

The first operating mode, which is purely numerical, is also the simplest. It takes the parameters of the system model, workload model, and the rates entered for the various UDE types, and from this estimates the UDE rate of the composed model and generates an exponentially distributed inter-arrival time. The internal clock which is used to determine the next event is then advanced to the indicated time, our metrics are updated, and we process the UDE as appropriate.

The second mode utilizes discrete event simulation to model the interaction of the workload with the faults injected into the storage system. In order to determine which events occurring in the storage system cause a user level undetected data corruption event, lists are maintained of both those chunks being actively read and written, and the chunks in the storage system which have suffered from a UDE. Only those events which add or remove chunks from the active list, or access faulty disk blocks are explicitly simulated. Since all other operations will either read a chunk composed of good blocks, or write a good block to a block which is already good, they cannot affect the state of blocks within the system. When the UDE can no longer manifest as a data corruption event, i.e. when all active chunks which intersect

with the faulty chunk are no longer active, or when the faulty chunk is overwritten with good data, we pass from discrete event simulation into our simulator's third operational mode.

The third mode is a hybrid of the first two, and combines discrete event simulation at a higher level of abstraction than the second mode, with calculations which utilize a rate adjustment to estimate the probability of a UDE before the next discrete event. Despite the fact that by the third mode of our solution technique the UDE can longer manifest we must be careful to respect the dependency the system has with respect to the last UDE injected into the system, for the admittedly unlikely case that a second UDE might occur before the system is no longer dependent on the events of the last UDE. This is done by modeling the flow of chunks into and out of the active list, and calculating the probability of a UDE before the next flow of a chunk into or out of the list and adjusting the rate of the next UDE based on the current composition of the active list. The following sections describe the modes in more detail.

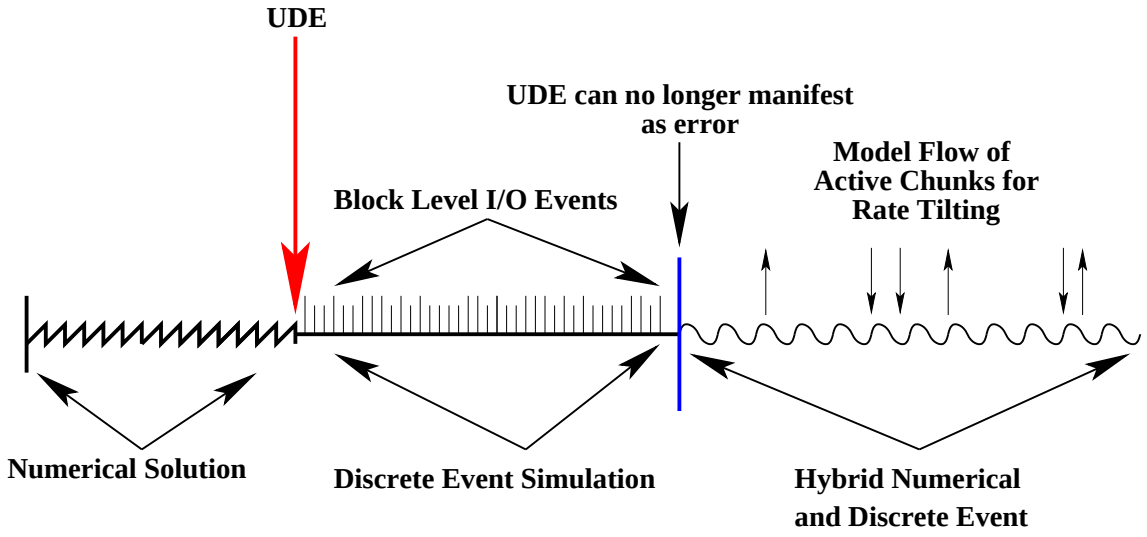


Figure 4.3: Simulator Architecture

4.4.1 Numerical Solution Mode

Our simulator begins in its first mode, and calculates the expected rate of a UWE or URE. This is done by first finding the expected number of reads and writes generated by an I/O operation to one of the RAID units. To show how this calculation is accomplished we show

the formulas for a RAID5 storage subsystem. Given a RAID5 strip size of $\text{size}_{\text{strip}}$, and a stripe width of w data disks plus one parity, the expected number of writes for a read-modify-write operation is equal to the number of required strips, plus one write per stripe written to update parity. One read is likewise required for each strip written in a row, plus an additional potential read if the write crosses a row boundary giving the expected number of writes for a read-modify-write as:

$$E[\text{writes/read-modify-write}] = \left\lceil \frac{E[\text{size}]}{\text{size}_{\text{strip}}} \right\rceil + \left\lceil \frac{\max(E[\text{size}], \text{size}_{\text{strip}}) - \text{size}_{\text{strip}}}{\text{size}_{\text{strip}}} \right\rceil + \left\lceil \frac{E[\text{size}]}{w \cdot \text{size}_{\text{strip}}} + \frac{\max(E[\text{size}], w \cdot \text{size}_{\text{strip}}) - \text{size}_{\text{strip}}}{w \cdot \text{size}_{\text{strip}}} \right\rceil \quad (4.1)$$

and the expected number of reads for a read-modify-write as:

$$E[\text{reads/read} - \text{modify} - \text{write}] = E[\text{writes/read-modify-write}]$$

On a read to the RAID5 system, only a single read is performed giving us the following reads and writes per read:

$$E[\text{writes/read}] = 0 \quad (4.2)$$

$$E[\text{reads/read}] = 1.0 \quad (4.3)$$

Using the values calculated with these equations it is possible to find the rate of UDEs, given an I/O rate of λ_{IO} of which p_{read} are reads, And it is possible, using these parameters to generate a time to the next UDE (T_{UDE}) as shown in the following equation:

$$T_{UDE} = 1/(\lambda_{IO} p_{read}(E[\text{writes/read}] + E[\text{reads/read}]) + \lambda_{IO}(1 - p_{read})(E[\text{writes/read-modify-write}] + E[\text{reads/read-modify-write}])) \quad (4.4)$$

Once we have determined a UDE will occur, we determine the type of UDE (UWE vs. URE) and the mechanism (dropped, near-off track, far-off track) via uniformization [51]. In the case of a URE, the error is transient, and we remain in numerical mode after handling the UDE. In the case of a UWE, the error does not manifest immediately, and we switch the simulator into the discrete event simulation mode.

4.4.2 Discrete Event Simulation Mode

The discrete event simulator makes only slight modifications to a standard discrete event simulator as, for example, described in [52]. Pseudo-code for a single pass through the main simulation loop is illustrated in Figure 4.4. The primary modification to the standard algorithms is our creation and maintenance of two lists, the active chunk list, C_A, and the tainted chunk list C_T. The set C_T is comprised of all chunks which have blocks affected by a UDE that can still serve corrupted data to the user. Between this set, and the set of all chunks being actively read and written, C_A, we can identify I/O events which do not affect system state and safely discard them without updating the state of the simulator beyond advancing the clock. This is because all events which do not modify the state of a block within C_T, or which do not alter the composition of C_A or C_T cannot serve corrupted or stale data, or affect the rate at which additional UDEs occur.

At each step in the simulator, we process one event, comparing it to C_A and C_T to see if our metric depends on the event, processing it if it does, and modifying C_A and C_T if necessary. We continue with discrete event simulation until such a time as $C_T \cap C_A = \emptyset$. When this occurs, all UDEs have either been mitigated, or the chunks they occurred on will not be read again until first written, and we no longer need to simulate I/Os on a block level as they cannot result in a data corruption error. However, it is not enough to simply return to our first, numerical, mode. So long as $C_{A,i} \neq \emptyset$ and $C_{A,i} \cap C_{A,0} \neq \emptyset$, where $C_{A,0}$ was the active chunk list when it was first true that $C_T \cap C_A = \emptyset$, and $C_{A,i}$ is the active chunk list at some time i, we have a dependence on the state of the simulator during the previous UDE, which must be maintained in the case that another UDE were to occur before $C_{A,i} = \emptyset$ or $C_{A,i} \cap C_{A,0} = \emptyset$, and thus we switch into our hybrid mode instead of the numerical mode.

```
nextEvent ← eventList.head()
clock ← clock + nextEvent.interarrivalTime
if nextEvent.io ∩(C_A ∪ C_T) then
  Process nextEvent
  if nextEvent reads a block which suffered from a UDE then
    Record UDE, update MTTDC
  else {nextEvent writes a set of blocks B which suffered from a UDE}
    C_T ← C_T ∪ B
  end if
end if
if nextEvent is a chunk leaving C_A then
  C_A ← C_A \ nextEvent
end if
if nextEvent.io \ C_A ≠ ∅ then
  C_A ← C_A ∪ nextEvent.io
  C_T ← C_T ∪ nextEvent.io
end if
Schedule new event of the same type as nextEvent
```

Figure 4.4: Pseudo-code representation of a single pass through the main loop of the discrete event simulator.

4.4.3 Hybrid Numerical and Discrete Event Mode

The Hybrid mode of operation of the simulator uses discrete event simulation to manage the contents of C_A. When a chunk leaves C_A, it removes it from the set, and when a chunk enters, it increments a counter of implicitly simulated active chunks, active chunks represented by this counter are treated as the expected value of an active chunk from the workload. In between events which modify C_A, the hybrid mode acts in a manner similar to the numerical mode described previously, calculating T_{UDE} as shown in Equation 4.4. Instead of using the parameters for the workload, however, it takes a less general approach and estimates the same parameters from C_A. It then uses these new estimates to perform a rate adjustment calculation for T_{UDE}, representing each of the parameters used to calculate T_{UDE} as a weighted average with the expected value of these parameters for the workload in general. The weights used are simply the fraction of active chunks in C_A as compared to the total number $|C_A| + implicit$. When $C_{A,i} = \emptyset$ or $C_{A,i} \cap C_{A,0} = \emptyset$, we return to the purely numerical scheme as we are no longer dependent on the last UDE scenario.

```
nextEvent ← eventList.head()
clock ← clock + nextEvent.interarrivalTime
if nextEvent is an addition to C_A then
  implicit ← implicit + 1
end if
if nextEvent is a chunk leaving C_A then
  if implicit > 0 and nextEvent ∉ C_A then
    implicit ← implicit - 1
  else
    C_A ← C_A \ nextEvent
  end if
end if
if C_A = ∅ then
  Switch to Numerical mode
end if
Adjust the rate for UDEs based on C_A and Implicit
Schedule next change in C_A
```

Figure 4.5: Pseudo-code representation of a single pass through the main loop of the hybrid discrete event simulator.

4.5 Example Model Solution

We simulated three different example systems to ascertain the effect of UDEs in three settings, a large-scale storage system, a large enterprise storage system, and a small business storage system. The large-scale system was modeled as a set 1000 disks each with a capacity of 1 terabyte, the enterprise system was modeled as a set of 512 disks, each with a capacity of 300 gigabytes, and the small business system was modeled as a set of 32 disks, also with capacities of 300 gigabytes each. Each of the three modeled systems were simulated with three different workloads, as discussed in Section 4.2. The rate of I/O operations per second for each system-workload pairing was calculated by multiplying the io/s parameter for the workload by the number of disks (excluding those used for parity) in the system. All systems, including those without mitigation, simulated a weekly parity scrub. A weekly scrub [53] was chosen based on the existing literature, which discusses a wide range of scrub intervals from every few days [54] to at least every two weeks [47, 9]. A scrub interval of one week was chosen as a representative rate, but is not integral to our simulator, and can be easily altered. Each of these systems was simulated with and without mitigation techniques.

Table 4.2: Estimated mean proportion of UDEs which manifest as undetected data corruption for various systems under the Abstract workload, with and without mitigation.

Mitigation	*Large System*	*Enterprise System*	*Small Business System*
No	0.718	0.718	0.718
Yes	0.0028	0.0028	0.0028

UDE rates of three different orders of magnitude $(10^{-11}, 10^{-12}, 10^{-13})$ were considered. A total of 10 million UDEs were simulated for each scenario.

4.5.1 Validation

The results of the simulator were validated using a simple combinatorial model of a UDE process. The combinatoric model functions the same as our simulator during the initial stage, but once it has decided a UDE has occurred, instead of switching to discrete simulation, it utilizes the DTMC described in Figure 4.1, using a geometric distribution with parameter equal to the transition probability to the opposite state to find the number of reads before a subsequent write event. After the UDE has been resolved, it simply calculates the next UDE, and continues as before. Comparing results for the proportion of UDEs manifesting as undetected data corruptions, and rates of user-level undetected data corruption errors from this combinatorial model with results from the simulator, with parity scrubbing and other mitigation features switched off, helped us to gain confidence in our simulator and ensure it was generating results which agreed with our combinatoric model. Estimation of rates at which UDEs manifested as user level undetected data corruption events by the simulator were within 4.6% of those given by our combinatoric model, for 10 million simulated UDEs with no scrubbing, and no mitigation.

4.5.2 Results

Tables 4.2 and 4.3 summarize the proportion of UDEs which actually manifest as an undetected data corruption event for the simulated scenarios. As can be seen in Table 4.2 the proportion of UDEs which manifest as an actual error does not vary significantly when varying system parameters. The various rates of UDEs/IO simulated are not listed as the

Table 4.3: Estimated mean proportion of UDEs which manifest as undetected data corruption for various workloads on the large system model, with and without mitigation.

Mitigation	*Abstract Workload*	*Read Heavy Workload*	*Write Heavy Workload*
No	0.718	0.275	0.887
Yes	0.0028	0.0011	0.0035

Table 4.4: Estimated mean and standard deviation of rate of UDE manifestation as undetected data corruption per second for various systems under the Abstract workload, with and without mitigation, for various rates of UDEs/IO.

		Estimated rate for various rates of UDEs/IO					
		10^{-11}		10^{-12}		10^{-13}	
System	**Mit.**	μ	0.95% conf.	μ	0.95% conf.	μ	0.95% conf.
Large scale	No	$6.278 \cdot 10^{-7}$	$\pm 3.26 \cdot 10^{-13}$	$6.282 \cdot 10^{-8}$	$\pm 3.87 \cdot 10^{-14}$	$6.282 \cdot 10^{-9}$	$\pm 3.29 \cdot 10^{-15}$
Large scale	Yes	$2.415 \cdot 10^{-9}$	$\pm 3.28 \cdot 10^{-14}$	$2.466 \cdot 10^{-10}$	$\pm 3.38 \cdot 10^{-15}$	$2.519 \cdot 10^{-11}$	$\pm 2.97 \cdot 10^{-16}$
Enterprise	No	$3.217 \cdot 10^{-7}$	$\pm 1.98 \cdot 10^{-13}$	$3.218 \cdot 10^{-8}$	$\pm 1.98 \cdot 10^{-14}$	$3.221 \cdot 10^{-9}$	$\pm 1.14 \cdot 10^{-15}$
Enterprise	Yes	$1.259 \cdot 10^{-9}$	$\pm 1.30 \cdot 10^{-14}$	$1.262 \cdot 10^{-10}$	$\pm 2.10 \cdot 10^{-15}$	$1.253 \cdot 10^{-11}$	$\pm 1.15 \cdot 10^{-16}$
Small	No	$2.012 \cdot 10^{-8}$	$\pm 8.30 \cdot 10^{-15}$	$2.012 \cdot 10^{-9}$	$\pm 5.77 \cdot 10^{-16}$	$2.012 \cdot 10^{-10}$	$\pm 8.27 \cdot 10^{-17}$
Small	Yes	$7.930 \cdot 10^{-11}$	$\pm 8.49 \cdot 10^{-16}$	$7.868 \cdot 10^{-12}$	$\pm 8.33 \cdot 10^{-17}$	$7.857 \cdot 10^{-13}$	$\pm 1.14 \cdot 10^{-17}$

results were the same for all three parameters. Adding the mitigation technique described in Section 3.3.5 had the effect of reducing the proportion of UDEs which manifested as undetected data corruption by two orders of magnitude. Table 4.3 shows the effect that varying the workload can have on the proportion of UDEs that manifest as an error. While both the Abstract workload and Write Heavy workload have similar rates of manifestation, the Read Heavy workload has a much lower proportion of UDEs which manifest as undetected data corruptions. In all cases, adding mitigation had the effect of reducing the proportion of UDEs manifesting as errors by two orders of magnitude.

Tables 4.4 and 4.5 illustrate the relationship between the rate of undetected data corruption errors and varying system and workload parameters, and rates of UDEs/IO. Table 4.4

Table 4.5: Estimated mean and standard deviation of rate of UDE manifestation as undetected data corruption per second for the large scale system under the all workloads, with and without mitigation, for various rates of UDEs/IO.

		Estimated rate for various rates of UDEs/IO					
		10^{-11}		10^{-12}		10^{-13}	
System	**Mit.**	μ	0.95% conf.	μ	0.95% conf.	μ	0.95% conf.
Abstract	No	$6.278 \cdot 10^{-7}$	$\pm 3.27 \cdot 10^{-13}$	$6.282 \cdot 10^{-8}$	$\pm 3.87 \cdot 10^{-14}$	$6.282 \cdot 10^{-9}$	$\pm 3.29 \cdot 10^{-15}$
Abstract	Yes	$2.415 \cdot 10^{-9}$	$\pm 3.28 \cdot 10^{-14}$	$2.466 \cdot 10^{-10}$	$\pm 3.38 \cdot 10^{-15}$	$2.519 \cdot 10^{-11}$	$\pm 2.97 \cdot 10^{-16}$
Read Heavy	No	$2.404 \cdot 10^{-7}$	$\pm 1.28 \cdot 10^{-13}$	$2.405 \cdot 10^{-8}$	$\pm 2.44 \cdot 10^{-14}$	$2.401 \cdot 10^{-9}$	$\pm 1.79 \cdot 10^{-15}$
Read Heavy	Yes	$9.345 \cdot 10^{-10}$	$\pm 1.31 \cdot 10^{-14}$	$9.310 \cdot 10^{-11}$	$\pm 1.14 \cdot 10^{-15}$	$9.476 \cdot 10^{-12}$	$\pm 1.68 \cdot 10^{-16}$
Write Heavy	No	$7.764 \cdot 10^{-7}$	$\pm 5.52 \cdot 10^{-13}$	$7.763 \cdot 10^{-8}$	$\pm 4.68 \cdot 10^{-14}$	$7.766 \cdot 10^{-9}$	$\pm 3.51 \cdot 10^{-15}$
Write Heavy	Yes	$3.048 \cdot 10^{-9}$	$\pm 2.39 \cdot 10^{-14}$	$3.014 \cdot 10^{-10}$	$\pm 2.03 \cdot 10^{-15}$	$3.038 \cdot 10^{-11}$	$\pm 4.09 \cdot 10^{-16}$

shows that the rate of UDEs manifesting as undetected data corruption declines both as the rate of UDEs/IO decreases, and as the size of the system decreases. Those scenarios which involved mitigation decreased the rate of undetected data corruption events by between two and three orders of magnitude. To put these rates into perspective, Table 4.6 shows the mean interval between undetected data corruption events that correspond to these rates. Table 4.5 summarizes the effect of varying the workload while holding the simulated system constant. We show the results for the large system only, as the results for enterprise and small business systems showed similar trends in the rates of UDE manifestation. The rates of manifestation vary in a manner consistent with Table 4.3 illustrating that workload does have an effect on the rate of UDE manifestation, but for the workloads and systems tested, varying the workload still yields a rate of undetected data corruption events within the same order of magnitude.

Table 4.6: Estimated mean interval between undetected data corruption events for all systems under the Abstract workload, with and without mitigation, derived from Table 4.4.

		Mean Interval		
System	**Mitigation**	10^{-11}	10^{-12}	10^{-13}
Large	No	18.43 days	184.2 days	5.04 years
Large	Yes	13.13 years	128.6 years	1258 years
Ent.	No	35.98 days	0.9854 years	9.844 years
Ent.	Yes	25.19 years	251.3 years	2531 years
Small	No	1.576 years	15.76 years	157.6 years
Small	Yes	399.9 years	4030 years	40358 years

4.6 Conclusions

Our results indicate that UDEs will continue to grow in their importance to system designers as storage systems continue to scale, a conclusion which seems affirmed out by field observation of UDEs in modern large scale storage systems. Even in the case of the small business system simulated, the rate of data corruption events without implementing mitigation is such that one would expect to see UDEs occurring in a population of such storage systems, and if we assume a UDE/IO rate of 10^{-11}, the rate is high enough to push the mean interval between undetected data corruption below the average lifetime of even a single such system.

Our results suggest that data scrubbing on a weekly schedule is not enough to reduce UDEs sufficiently, as the data scrub must be processing the location on a disk where the UDE has occurred after the time at which the disk suffered the fault, and prior to the next read request for that location. It is particularly telling that when the rate is set at 10^{-12} UDEs/IO, the rate of UDEs/IO estimated for near-line disks, both enterprise and large scale systems have rates of undetected data corruption which place the mean interval between such events at less than a year. Given the growing practice of utilizing cheaper near-line drives in a RAID configuration, this highlights an important limitation of RAID. Since RAID5 and RAID6 can only detect errors on the data drives during a parity scrub, the protection they implement is largely orthogonal to the issues posed by UDEs. Fortunately, our simulations suggest that by using relatively simple techniques such as the sequence number approach to mitigation modeled in this work and described in [40], UDEs can largely be eliminated for the expected lifetime of near-future systems. While they double the requirement for read operations, requiring reading both the data and the parity, these methods reduce the rate of undetected data corruption events by two to three orders of magnitude.

We consider in this chapter only a single mitigation technique, and a set of relatively simple RAID based systems. Additionally, it is difficult to evaluate the correctness of our measures given that our multi-modal simulator utilizes a variety of techniques for solution, and only systems which can be trivially decomposed can be solved with these methods. In Chapter 5 we will outline a modeling language for describing storage systems, and method for identifying and categorizing complex dependence relationships within our models. Combined with information about the fault and recovery events within a given system, we will propose an algorithm for solving these models using a hybrid simulator which is more general than the one defined in this Chapter. In Chapter 6 we will expand our model to include all of the faults defined in Chapter 3.

CHAPTER 5

ANALYZING DEPENDENCE RELATIONSHIPS IN STORAGE SYSTEMS

5.1 Introduction

In Chapter 2, we introduced methods for building component-based models of storage systems and simulating them using empirical data. In Chapter 3, we discussed classes of faults that were less significant in smaller systems, but which have become relevant to modern systems as they continue to increase in scale. Chapter 4 presented an initial attempt to model those faults in component-based models when the models themselves are easily decomposable. The effect those faults have on storage systems is often more complex, especially when considered along with additional complications, such as data deduplication. In this chapter, we will detail a method for analyzing more complex models that cannot be trivially decomposed, and that represent larger more complex systems.

Like all large and complex systems, the systems we address present significant problems for modelers. Large systems present challenges for numerical solution, and though simulation can be used even for infinite state spaces, solution time grows with the number of events that must be processed. Because the faults we have defined have rates many orders of magnitude smaller than those of our I/O processes and mitigation methods, we will call them *rare events*. Models that contain rare events are often deemed stiff, meaning that for the estimates of the chosen metrics to converge, an increasing number of events must be observed. Given the rarity of UDEs and LSEs, rare events are a particular problem for large-scale storage systems.

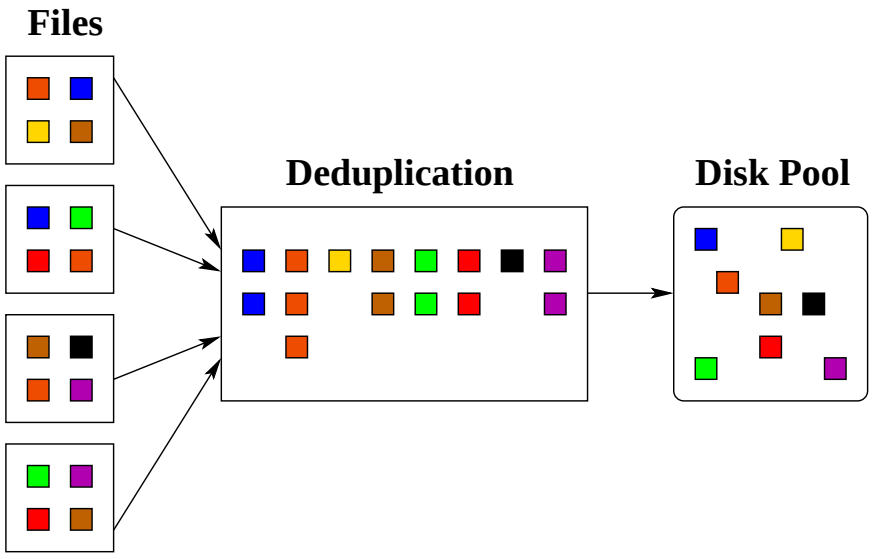


Figure 5.1: Representation of a deduplicated storage system. Redundant data is identified and eliminated, so that only unique data is stored.

5.2 Dependence in Storage Systems

Modern storage systems are often more complex than the simple system represented in Chapter 4. Most modern systems utilize RAID to improve their fault tolerance. In these cases, dependencies are created between disks in a RAID group. Parity drives have state that depends on the states of other disks within the RAID group; upon a failure, the states of all drives in the group are needed in order to re-create the lost drives.

Modern storage systems have begun using a technique known as *deduplication* to improve storage efficiency. The growing use of deduplication introduces a new form of dependence, which can potentially exist between any two disks in a deduplicated system. Deduplication (Figure 5.1) operates by fingerprinting data that is being written to the storage system and identifying sub-file chunks that already exist in the storage system, selecting those files for deduplication. When a file is deduplicated, only a single copy is stored as a deduplicated instance; a series of references to that instance are created for the additional copies identified. That creates dependence among the disks storing references, the disk storing the deduplicated instance, and the other disks in the RAID group that stores the instance.

Data deduplication is increasingly being adopted to reduce the data footprint of backup

and archival storage, and more recently has become available for near-line primary storage controllers. Scale-out file systems are increasingly diminishing the silos between primary and archival storage by applying deduplication to unified petabyte-scale data repositories spanning heterogeneous storage hardware. Cloud providers are also actively evaluating deduplication for their heterogeneous commodity storage infrastructures and ever-changing customer workloads. A more detailed model of deduplication based on empirical data will be presented in Chapter 6.

While these dependencies make models of the resulting storage systems necessarily more complex than models of storage systems with no fault tolerance or data deduplication, we note that these dependencies only effect metrics in certain situations. When concerning ourselves with reliability analysis, we note that these dependencies can be ignored except in the cases that a fault has occurred, causing the loss of a disk in a RAID unit, or the loss of a referenced deduplicated instance. Given that these failures are several orders of magnitude more rare than non-faulty events within the storage system, such as scrubbing, and I/O, an assumption that the disks are in fact independent holds for much of the storage systems lifetime. In this chapter we focus on ways to exploit this insight to achieve performance gains when modeling storage systems.

5.3 Related Work

There has been limited work in the literature which attempts to model complex storage systems in rich fault environments. In Chapter 6 we will discuss the state-of-the-art, and why it has been insufficient to understand the effect of deduplication on fault-tolerance. In this chapter, we present related modeling work from other domains that attempt to tackle similar modeling and solution difficulties in more general systems as preparation for introducing our framework and methods.

Rare events, despite the low probability of their occurrence, can have large impacts. As systems increase in size and complexity, rare event failures pose increased risk when they occur on a per-component basis. While such failures may still occur with the same proportion to other events, large-scale HPC systems can be expected to suffer a number of rare events

within their normal lifetimes [55]. Rare faults may also be important for safety-critical systems in which they are still rare during the system life time as they can represent faults which cause catastrophic data loss [40] or system failure.

Importance sampling attempts to reduce the variance of estimates of a model's reward variables through mathematical biasing of the simulation, increasing the proportion of rare events witnessed. This is accomplished by biasing the distributions, yielding a biased estimator. An appropriate must then be found to unbias the estimators [56, 57]. While importance sampling can speed-up simulations of models that have rare events, choosing a set of biased distributions and unbiasing functions for the estimators can prove difficult. Improper choices may slow down the simulation, or worse (and perhaps more likely), yield incorrect estimators for the performance variables.

Importance splitting biases the simulation to make rare events less rare using a different approach. The state space is partitioned into a number of *levels*. A level contains those states that form critical points on "important paths" in the simulation, i.e. which result in an increased probability of witnessing a rare event. Simulation paths that reach a level are split and re-sampled to increase the likelihood of witnessing a rare event. Trajectories resulting from a split are correlated, generating an unbiased estimator of the variance is not straightforward. Selection of appropriate points to split and levels are model-specific problems which can impact the efficiency of solutions using this technique [58, 59].

Decomposition offers an approach for large, complex systems, dividing them into smaller submodels and finding solutions for the submodels separately. Models which cannot be broken into wholly independent submodels must also have their interactions characterized. If the submodels are weakly coupled, they may be sometimes be considered as a composition of nearly independent submodels [60, 61].

5.4 Modeling Preliminaries

We present our method in the context of a generic model specification language based on the notation presented in [62]. This is intended as an alternative to presenting our results in a specific formalism, both to simplify the discussion of our techniques and to generalize

our methods.

Definition 1. *A model is a 5-tuple* $(S, E, \Phi, \Lambda, \Delta)$

- S *is a finite set of state variables* $\{s_1, s_2, \ldots, s_n\}$ *that take values in* $\mathbb{N}$.
- E *is a finite set of events,* $\{e_1, e_2, \ldots, e_m\}$ *that may occur in the model.*
- $\Phi : E \times \mathbb{N}_1 \times \mathbb{N}_2 \times \ldots \times \mathbb{N}_n \to \{0, 1\}$ *is the event-enabling function specification. For each event* $e \in E$*, and any set of state variables and their assignments,* q *(represented by an ordered set of natural numbers), event* e *is enabled and may occur for this set of state variable assignments iff* $\Phi(e, q) = 1$.
- $\Lambda : E \times \mathbb{N}_1 \times \mathbb{N}_2 \times \ldots \times \mathbb{N}_n \to (0, \infty)$ *is the transition rate function specification. For each event* e*, and set of state variables and their assignments,* q*, event* e *occurs with rate* $\Lambda(e, q)$ *when the state variables of the model have the values given in* q.
- $\Delta : E \times \mathbb{N}_1 \times \mathbb{N}_2 \times \ldots \times \mathbb{N}_n \to \mathbb{N}_1 \times \mathbb{N}_2 \times \ldots \times \mathbb{N}_n$ *is the state variable transition function specification. For each event* $e \in E$*, and each set of state variables and their assignments* $q \in \mathbb{N}_1 \times \mathbb{N}_2 \times \ldots \times \mathbb{N}_n$*,* $\Delta(e, q) \to q'$ *defines the values assigned to all state variables of the model when* e *occurs.*

Definition 2. *The state of a model* M *is a mapping* $\psi : S \to \mathbb{N}$*, where for all* $s \in S, \psi(s)$ *is the value of the state variable* s*.* $\Psi = \{\psi | \psi : S \to \mathbb{N}\}$ *is the set of all such mappings.*

A trajectory, or behavior of a model, is described as a finite sequence of states and events. The model is assumed to be in some initial state, with events occurring with a rate determined by λ. When an event fires, the model transitions in accordance with the state transition function δ. The probability of transitioning from some arbitrary state, ψ_i, to a particular next state, ψ_j is the probability that an event e is the next event such that $\delta(\psi_i, e) = \psi_j$. We calculate this probability as:

$$P(\psi_i \to \psi_j) = \frac{\sum_{e \in E | \delta(e, \psi_i) = m_j} \lambda(e, \psi_i)}{\sum_{e \in E | \phi(e, \psi_i) = 1} \lambda(e, \psi_i)}. \tag{5.1}$$

In addition to specifying a model of system, one must specify the performability, availability, or dependability measures for a model. These measures are specified in terms of reward variables [63]. Reward variables are specified as a reward structure [64] and a variable type.

Definition 3. *Given model $M = (S, E, \Phi, \Lambda, \Delta)$, we define two reward structures: rate rewards and impulse rewards.*

- *A rate reward is defined as a function $\mathcal{R} : \mathcal{P}(S, \mathbb{N}) \to \mathbb{R}$, where for $q \in \mathcal{P}(S, \mathbb{N})$, $\mathcal{R}(q)$ is the reward accumulated when for each $(s, n) \in q$ the marking of s is n.*
- *An impulse reward is defined as a function $\mathcal{I} : E \to \mathbb{R}$, where for $e \in E, \mathcal{I}(e)$ is the reward earned upon completion of e.*

where $\mathcal{P}(S, \mathbb{N})$ is the set of all partial functions between S and $\mathbb{N}$.

Definition 4. *Let $\Theta_M = \{\theta_0, \theta_1, \ldots\}$ be a set of reward variables, each with reward structure $\mathcal{R}$ or $\mathcal{I}$ associated with a model M.*

The type of a reward variable determines how the reward structure is evaluated, and can be defined over an interval of time, an instant of time, or in steady state, as shown in [65, 63].

5.4.1 Instant-of-Time Variables

We refer to variables which are used to measure the behavior of a model at a particular time t as instant-of-time variables [66, 63]. Such a variable, $\theta(t)$ is defined as:

$$\theta_t = \sum_{\nu \in \mathcal{P}(S,\mathbb{N})} \mathcal{R}(\nu) \cdot I_t^{\nu} + \sum_{e \in E} \mathcal{I}(e) \cdot I_t^{e} \tag{5.2}$$

where

- I_t^{ν} is an indicator random variable which represents the instance of a SAN in a marking such that for each $(s, n) \in \nu$, the state variable s has a value of n at time t.

- I_t^e is an indicator random variable which represents the instance of an event e that has fired most recently at time t.

If the indicator random variables converge in distribution as $t \to \infty$, we consider θ_t to be in "steady state". We can then evaluate the steady state reward at an instant of time t with the variable defined as

$$\theta_{t\to\infty} = \sum_{\nu\in\mathcal{P}(S,\mathbb{N})} \mathcal{R}(\nu)\cdot I^{\nu}_{t\to\infty} + \sum_{e\in E}\mathcal{I}(e)\cdot I^{e}_{t\to\infty} \tag{5.3}$$

where

- $I^{\nu}_{t\to\infty}$ is an indicator random variable which represents the instance of a SAN in a marking such that for each $(s,n)\in\nu$, the state variable s has a value of n in steady state.
- $I^{e}_{t\to\infty}$ is an indicator random variable which represents the instance of an event e that has fired most recently in steady state.

5.4.2 Interval-of-Time Variables

In order to calculate metrics which accumulate over some fixed interval of time, we use interval-of-time variables. Such variables accumulate reward during some interval of time, and take on the value of the total reward for the defined period [66, 63]. Given such a variable, $\theta_{[t,t+l]}$, we define it as:

$$\theta_{[t,t+l]} = \sum_{\nu\in\mathcal{P}(S,\mathbb{N})} \mathcal{R}(\nu)\cdot J^{\nu}_{[t,t+l]} + \sum_{e\in E}\mathcal{I}(e)\cdot N^{e}_{[t,t+l]} \tag{5.4}$$

where

- $J^{\nu}_{[t,t+l]}$ is a random variable which represents the total time the model spent in a marking

such that for each $(s, n) \in \nu$, the state variable s has a value of n during the period $[t, t + l]$.

- $I^e_{t\to\infty}$ is an random variable which represents the number of times an event e has during the period $[t, t + l]$.

5.4.3 Time-Averaged Interval-of-Time Variables

The final type of reward variables we will consider are time-averaged interval-of-time variables. These variables quantify accumulated reward averaged over some interval of time [66, 63]. Given such a variable, $\theta'_{[t,t+l]}$ we define it as:

$$\theta'_{[t,t+l]} = \frac{\theta_{[t,t+l]}}{l} \tag{5.5}$$

5.5 Dependence in Storage Systems

In Chapter 6 we will attempt to develop a good understanding on the reliability of systems which use deduplication, which has to date been a difficult task for system designers [22]. Deduplication itself poses two potential reliability problems. First as illustrated in Figure 5.2a, the fact that chunks in several files depend on a single instance means that loss of a instance causes secondary losses across the data store. Inversely, as shown in Figure 5.2b, files with multiple references to different chunks may be dependent on the reliability of multiple storage devices. If any one of the devices upon which it depends fails, the file itself is lost. This additional dependence may be counter-balanced, however, depending on the degree of additional storage efficiency. By decreasing the number of disks in the system with deduplication, we decrease the number of expected disk faults as well. Understanding these complex relationships requires understanding the nature of deduplication itself, as well as the complex interactions created by the underlying storage system.

In order to properly exploit the dependence relationships present in a modeled storage system, we must first establish an understanding of the types of relationships in which we

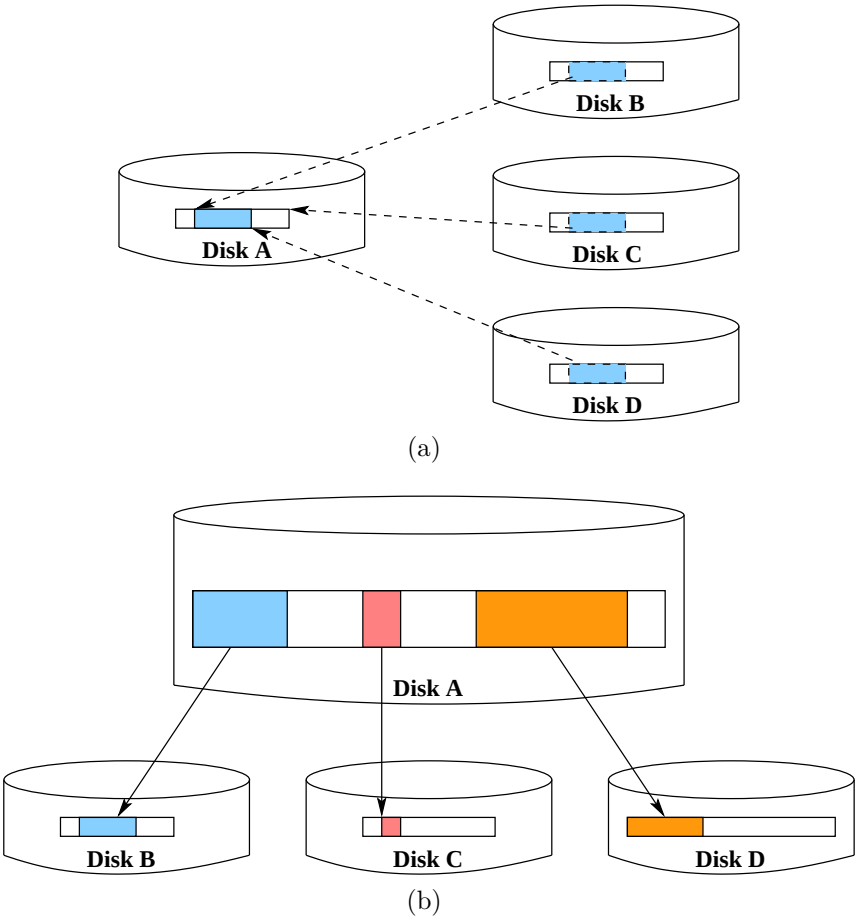


Figure 5.2: Deduplication example showing the dependence of multiple references to the same chunk in multiple files to a single stored reference and in a file to multiple disks in the data store.

are interested. Our goal is to identify and exploit structural properties in such a model, primarily failures due to disk failure, LSEs or UDEs, and recovery actions which remove these failures. Our hypothesis is that failures create important dependence relationships within the model, causing us to evaluate otherwise independent submodels as a larger model. We hypothesize that repair actions break these dependencies, until the next failure occurs. Furthermore, we are interested in failure and repair actions that represent rare-events.

5.5.1 RAID-Induced Dependence

Under normal operating conditions, the components of a storage system can be considered largely independent. Given the assumption of uniform file placement on the system (as is typical for large-scale general purpose machines), files are read and written to drives in an independent fashion. In such cases the system might be modeled more tractably as a set of independent systems, with each system solved individually. The use of RAID, however, implies a dependence between some sub-systems in the case of a failure. Should two drives suffer partial or total failure, they can no longer be treated as independent if they belong to the same RAID group, until the failure is repaired.

Once a failure has occurred, the entire RAID group must be considered dependent as further failures directly impact the integrity of files in the RAID group. This dependence can be removed once successfully recover actions have repaired all failures within the RAID group, allowing the disks in the group to once again be considered independent.

5.5.2 Deduplication Induced Dependence

An additional form of dependence in storage systems are those caused by deduplication. When a failure occurs for a chunk which stores an instance of a deduplicated resource in the storage system, a dependence is created for all references to that chunk, and the other disks in the failed instances RAID group. Should the RAID suffer additional failures which make the instance unrecoverable, all references to the instance will also be unrecoverable. As before, recovery of the failed instance and repair eliminates this dependence.

5.5.3 Important Events

For both of the major sources of dependence, an important point is that the events that cause and remove dependence are rare events. Faults of interest occur rarely in the system, because of the rates used by the models that represent them. Repair actions, while having rates that are relatively fast compared to that of failures, can only occur when a fault has changed system state. Thus, they are rare due to their enabling conditions, which are rarely met. In later sections, we will analyze rare events along with dependence relationships to form a strategy for solving our systems.

5.6 Understanding Dependence Relationships

The dependence relationships described for storage systems, using RAID and deduplication seem similar, in concept, to the idea of *near-independence* [67]. In order to better characterize the dependence relationships present in models of storage systems we discuss notions of dependence used in studies of near-independence. Characterizing the dependence of multiple portions of a model is complex, and involves development of a measure of how far the model is from an ideal set of truly independent models. For that reason, using the terminology discussed in [67], we will simply present a qualitative discussion of structures that can arise.

We represent our formalism, in Figure 5.3a, using circles for state variables and squares for events. Dependencies are shown using directed arcs. In Figure 5.3a, we have a model with state variables s_0, s_1, s_2, and s_3 and events e_0, e_1, e_2, and e_3. While the state of the submodel constructed from state variables s_0, s_1 does not depend directly on events fired by the second submodel (which is constructed of state variables s_2, s_3), it can still depend on the state of the other submodel in two key ways.

- *Rate dependence*: The two submodels can be said to have *rate dependence* if the transition rate function Λ of an event in one submodel is defined in terms of the state variables in the other submodel. For example, we might define the rate function of e_1

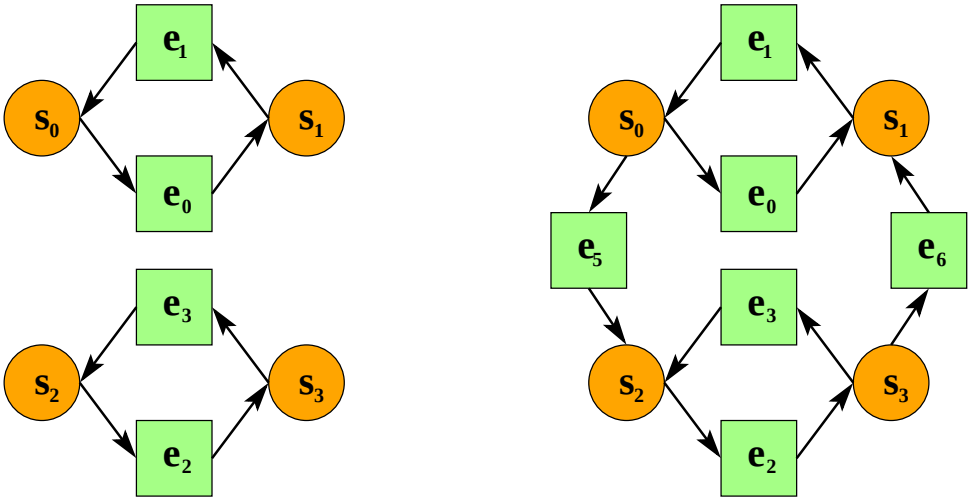


Figure 5.3: Two examples of near-independent submodels.

in Figure 5.3a as follows.

$$\begin{aligned} \Lambda(e_1, s_2 = 0) &\rightarrow x \\ \Lambda(e_1, s_2 \neq 0) &\rightarrow y \end{aligned}$$

Even though the firing of events in a given submodel cannot affect the state variables of another submodel, the rate of the event of one submodel is dependent on the state variables of another submodel.

- *External dependence*: When an event in one submodel has an event-enabling function, Φ, defined in terms of the state or state variables of another submodel, we say the submodels feature *external dependence*. For example, we might define the rate function of e_1 in Figure 5.3a as follows.

$$\begin{aligned} \Phi(e_1, s_2 < x) &\rightarrow 0 \\ \Phi(e_1, s_2 \geq x) &\rightarrow 1 \end{aligned}$$

Again, even though the firing of events in a given submodel cannot affect the state variables of another submodel, e_1 cannot fire, unless the other submodel has $s_2 \geq x$.

The enabling conditions of an e_1 are dependent on the state of s_2.

A third type of structure, synchronization dependence (which we call Δ-dependence, is discussed in [67]. It corresponds to simultaneous changes in the values of two or more state variables in two or more submodels. We expand upon this notion in light of our concern with rare faults to describe a new structural feature, illustrated in Figure 5.3b.

- Δ-*dependence*: When the firing of an event changes the value of state variables in two or more otherwise independent submodels, we say that they feature Δ-*dependence*.

While the submodel shown in Figure 5.3b would not normally be considered near-independent, if e_4 and e_5 represent rare events, the states of the two submodels depend on each other only rarely, when those events fire. Assume that the initial state of the model is such that $s_0 = s_2 = 1$, and $s_1 = s_3 = 0$. Assume that the rates of the events are not state-dependent, and that events are enabled when a state variable for which they have an incoming arc is greater than zero, and disabled otherwise. Also assume that the rates of e_5 and e_6 are several orders of magnitude more rare than e_0, e_1, e_2 and e_3. Until e_5 or e_6 fires, the model in Figure 5.3b will behave just like the model in Figure 5.3a under the same initial conditions and assumptions about the events. However, once either e_5 or e_6 fires, the model will change so that one submodel is disabled and inactive with $s_2 + s_3 = 0$, while the other has $s_0 + s_1 = 2$. This will remain true, and the two submodels will be independent of one another, until either e_5 or e_6 fires again, returning the system to one that resembles 5.3a again, with $s_0 + s_1 = 1$ and $s_2 + s_3 = 1$.

Dependencies between submodels result from *direct dependencies* between events and states, or from *indirect dependencies* resulting from a sequence of direct dependencies. Each direct dependence can be classified as a rate dependence, an external dependence, or a Δ-dependence.

Recall from Definition 3 in Section 5.4 that a reward variable may have one of two reward structures, rate reward or impulse reward; this implies a final category of dependence.

- *Reward dependence*: When a reward variable $\theta_i \in \Theta_M$ exists such that its reward structure is defined in terms of the state variables of two submodels, or in terms of the events of two submodels, we say they feature reward dependence.

Unlike the other forms of dependence we have defined, reward dependence is more likely to inhibit decomposition of our model into submodels, for reasons that will become apparent in Section 5.8.

5.6.1 Model Dependency Graph

Now that we have identified general ways in which submodels can be related, we define a way to codify these relationships by constructing a *model dependency graph* (MDG). We will use an MDG in conjunction with rare events found in the model via the methods discussed in Section 5.7 as inputs to an algorithm we introduce in Section 5.8, to provide a proposed decomposition of M.

Definition 5. *The MDG of a model M is defined as an undirected labeled graph, $G_M = (V, A, L)$, where V is a set of vertices composed of three subsets $V = V_S \cup V_E \cup V_\Theta$, A is a set of arcs connecting two vertices such that one vertex is always an element of the subset V_S and one vertex is always an element of the subset V_E, or one vertex is of the subset V_Θ while the other is of the set $\{V_E \cup V_S\}$, and L is a set of labels applied to elements of A from the set $\{\Phi, \Lambda, \Delta, R\}$. Let V_S denote the subset of vertices representing the state variables $S \in M$; V_E denote the subset of vertices representing the events $E \in M$, and V_Θ denote the subset of vertices representing reward variables from Θ_M.*

We construct G_M using the model specification from Definition 1 of a model M, from Section 5.4. G_M has a node for every state variable in S and event in E, and reward variable in Θ_M, with arcs connecting an arbitrary state variable s_i to an arbitrary event e_j, iff

- The enabling condition of e_j depends on the value of s_i. This indicates an *external dependence* and is marked with the label Φ.
- The rate of the event e_j depends on the value of s_i. This represents a *rate dependence* and is marked with the label Λ.
- The firing of e_j changes the value of s_i. This represents a Δ *dependence* and is marked with the label Δ.

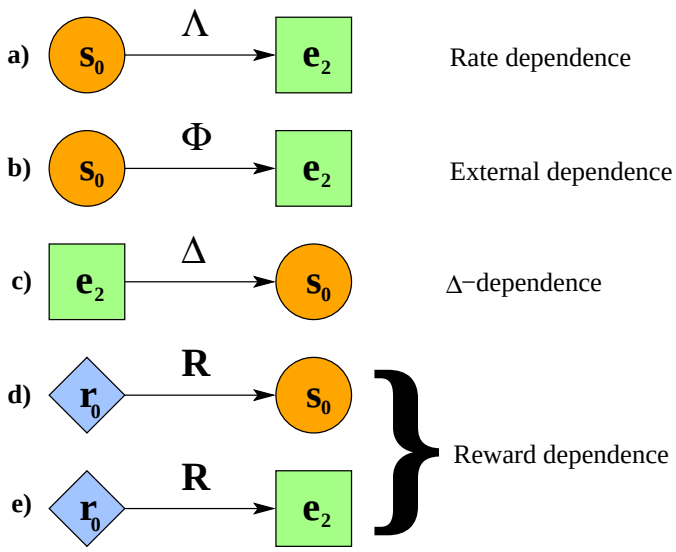


Figure 5.4: Two examples of near-independent submodels.

An arc labeled R connects a node representing an element $\theta_i \in \Theta_M$ to a node representing an element $a_j \in \{S \cup E\}$ iff

- $a_j \in S$ and θ_i is a rate reward defined in terms of a_j.
- $a_j \in E$ and θ_i is an impulse reward defined in terms of a_j.

We represent an MDG graphically as shown in Figure 5.4. State variables are represented by circles, events by squares, and reward variables by diamonds. Arcs in an MDG represent dependencies. As shown in Figure 5.4 rate dependencies are labeled Λ (a); external dependencies are labeled with Φ (b); Δ-dependencies are labeled with Δ (c); and rate dependencies, whether impulse or rate rewards, are labeled with R (d,e).

Proposition 1. *For a given model M, the graph G_M represents all possible dependencies between all events and state variables in a model.*

Proof. Proof by contradiction. Suppose there exist some state s_i and some event e_j that are directly dependent and not captured by G_M. All direct dependencies due to Φ, Λ, and Δ are encoded in G_M as labeled edges thus, the dependence must be one outside of the definition of Φ, Λ, and Δ. By definition 1, no such direct dependencies can exist. Thus our graph represents all possible direct dependencies.

Suppose there exist two elements $\alpha, \beta \in S \cup E$ that are indirectly dependent and not captured by the graph G_M. They are indirectly dependent if they are both state variables and the value of α can affect the value of β or vice versa. If α is a state variable and β is an event, they are indirectly dependent if the firing of β can affect the value of α, or if the value of α can affect the value of $\Phi(\beta, q), \Lambda(\beta, q)$, or $\Delta(\beta, q)$ for $q \in \mathbb{N}_1 \times \mathbb{N}_2 \times \ldots \times \mathbb{N}_n$. If α and β are both events, they are indirectly dependent if the firing of α can affect the value of $\Phi(\beta, q), \Lambda(\beta, q)$, or $\Delta(\beta, q)$ and vice versa.

In the model this indirect dependency will take the form of a series of event firings and state variable changes, each of which is either enabled by, or has its rates set by, a state variable upon which it depends, and which changes the value of subsequent state variables upon which future events depend. For such a sequence $\{s_i, e_j, s_k, e_l, \ldots\}$ to exist, every consecutive pair in the sequence $(s_i, e_j), (e_j, s_k), (s_k, e_l), \ldots$ must be directly dependent. If this is true, then there must exist a path defined by a series of vertices in V and arcs in A from the vertex representing the starting state or event in the sequence, to the vertex representing the final state or event in the sequence, such that path visits each vertex that corresponds to intermediate states and events in the sequence. Therefore the indirect dependence of α and β must be represented by the path of direct dependencies $v_\alpha, v_\alpha v_i, v_i, \ldots, v_j, v_j v_\beta, v_\beta$, and thus cannot exist. □

5.7 Rare Events in Storage Systems

The dependence relationships of interest are those which become important when a fault occurs, and can be ignored again when a fault is repaired. In both instances, faults, and repairs, the action which changes the importance of the dependence relationship can be classified as a rare event. In the case of failures, the event is rare because its rate is very low, compared to other events in the system. In the case of repair actions, the event is rare because it is only enabled when a fault has occurred in the system.

In order to find a way to decompose a storage model, we could require the user to specify the failure and repair actions in the model, as with the decomposition methods presented by [67]. Ideally, however, we wish to be able to identify these events without user input. The

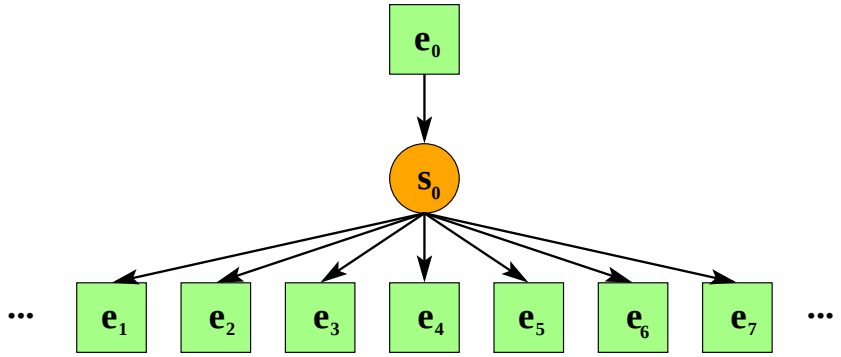


Figure 5.5: Models exhibiting rare events due to competition.

characteristics that set these events apart from others in the model is that they are rare.

5.7.1 Identifying Rare Events

In order to find the events which represent whole disk faults, LSEs and UDEs, we need to identify events which are "locally rare." An event e_i, $\Lambda(e_i, q)$ may be defined such that it's rate is much less than that of other events in the model, i.e. $\Lambda(e_i, q) < \mu_{max} \forall q$. In these cases we can classify the local rate of e_i to be rare. In the case of an event with a state-dependent rate (i.e., where $\Lambda(e_i, q)$ varies for different q), it may be useful to create two virtual events, $e_{i,1}$ and $e_{i,2}$, with the first virtual event replacing e_i for values of $\Lambda(e_i, q)$ that constitute non-rare events, and $e_{i,2}$ replacing e_i for values that qualify as representing rare events.

For our deduplication system, these locally rare rates which have $\Lambda(e_i, q) < \mu_{max} \forall q$, play a part in identifying rare events in the case of total disk failures, initial latent-sector errors, and undetected disk errors. These events have rates which are rare compared to others within the model, based simply on the evaluation of their rate function $\Lambda(e_i, q)$.

Figure 5.5 illustrates an example in which the local rates defined by the transition rate function does not differentiate rare events from non-rare events. Assume that the rate of the event labeled e_0 is defined as $\Lambda(e_0, q) = \mu, \forall q$, and that the events labeled e_1, e_2, e_3 have rates defined as $\Lambda(e_i, q) = \mu, \forall q$ as well. Considering the case when the enabling function is

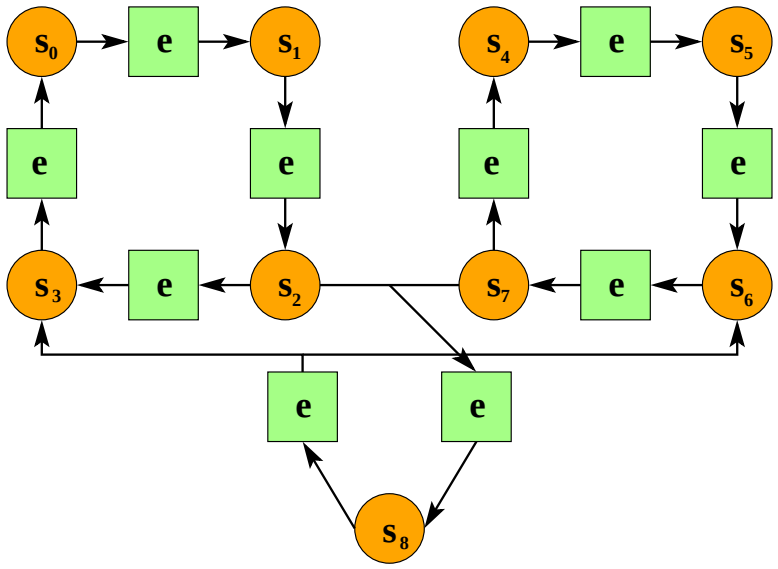


Figure 5.6: Models exhibiting rare events due to rare enabling conditions.

defined for all events except e_0 as

$$\Phi(e_i, s_0) = \begin{cases} 1, & \text{if } s_0 > 1 \\ 0, & \text{otherwise} \end{cases}$$

and the state transition function is defined in part by $\Delta(e_i, (s_0 = 1)) = (s_0 = 0)$.

If we imagine a similar case in which n such events are in competition, their effective rates might be much lower than the local rates defined in Λ would imply. The effective rate of each event can be easily determined using uniformization.

The final, and potentially most difficult to identify, fashion in which events may be rare is when their enabling conditions defined by Φ are rare. Despite the difficulty in finding such events, they are important as they represent recovery/repair actions, among other things. Consider the model presented in Figure 5.6. Assume that all events labeled either e or r have the same transition rate function, and that the model begins with state variables s_0 and s_4 equal to one, with all other state variables equal to zero. An event is enabled when all state variables with outgoing arcs pointing to the event have values greater than zero, and disabled otherwise. When an event fires, it decrements by one all state variables with outgoing arcs at the event, and increments by one all state variables with incoming arcs.

Although the rates of all events are similar, the enabling conditions of the events labeled r

are true far less often than those of the events labeled e. They require that the submodels be "synchronized," i.e., that $s_2 = s_7 = 1$ in order to fire. The enabling condition for the second r requires that $s_8 = 1$, a condition only true after the firing of a rare event, and before any other events have been fired. These events are rare because their enabling conditions depend on a model state that is rare.

It is important to identify events representing recovery, mitigation, and propagation as well as faults. The difficulty is that recovery actions have high rates compared to most failures. However, since they are not enabled unless a failure has occurred, they are dependent on a rare enabling condition. Latent sector errors also partially fall into this category. While an initial LSE is defined by a locally rare rate, studies [10] have shown that there is a period afterwords during which they become frequent. This precondition of a recent LSE creates a condition where an otherwise common event, is rare due to the state in which it is common being rare.

We combine these notions of how an event might qualify as rare by calculating the effective global rate of an event. For an event e_i we solve for the global rate μ_{e_i}, given a specification of the form presented in Definition 1 and Definition 2 from Section 5.4, along with π^*, the steady-state occupancy probability vector, as follows:

$$\mu_{e_i} = \sum_{\forall \psi_j | \phi(e_i, \psi_j)=1} \lambda(e_i, \psi_j) \left(\frac{\lambda(e_i, \psi_j)}{\sum_{\forall e \in E | \phi(e, \psi_j)=1}} \right) \cdot \pi^*[\psi_j] \tag{5.6}$$

While λ and ϕ in equation 5.6 are given by the model definition, π^* is not. In fact, solving for π^* can be difficult, potentially as difficult as solving for the original model itself.

5.7.2 Identifying Recovery Actions

While calculation of all rare enabling conditions is a difficult problem, we can use domain knowledge of storage system reliability models to aid us in finding certain classes of rare enabling conditions. As we noted before, recovery actions are by necessity paired with a previous fault in the model. Without a fault, the state variables in the model can not have values such that the recovery action can fire, and so recovery actions directly depend on rare

events. By analyzing the dependence relationships in our model we can identify these rare events given the assumption that our model begins with no initial faults.

In the next section we will introduce a method for decomposing models with rare events. Using the algorithms introduced in Section 5.8 we will show how to identify recovery, mitigation, and fault propagation events.

5.7.3 Partitioning

We identify a certain subset $E_R \subset E$ as rare events, given some partitioning scheme, as first discussed in Section 5.7.1. Choice of a static parameter with which to partition has been well-studied for hybrid simulation [68]. Some algorithms even propose methods for dynamic partitioning while simulating a given system [69]. The exact choice of partitioning method is unimportant for the correctness of the general application of our technique, but some approaches may have advantages when applied to specific models. In general, for the models we have studied, it has been appropriate to assume a static partitioning parameter $\mu_{\max}$. In practice, choice of a value for $\mu_{\max}$ is simple, as fault events are many orders of magnitude rarer than non-fault events. Two clusters of rates were easily identified using k-means clustering with two clusters, for the fault models defined in Chapter 3 and Chapter 6. Given a value for $\mu_{\max}$, we find those events $e_i \in E$ for which $\Lambda(e_i, q) < \mu_{max} \forall q$ and define the set of these events as E_R. This partitioning identifies those rare events that are rare due to a locally rare rate, or competition; it will allow us to identify those rare events that represent mitigation and fault propagation using the methods we will present in Section 5.8.

5.8 Decomposing Models with Rare Events

Using an MDG, we have shown how to represent all dependencies in a model of a storage system, and we have identified a subset of rare events within the model, which include failures and repair events. In this section we present an algorithm for decomposing M, using the MDG generated from M, G_M. M will be decomposed into a set of n submodels

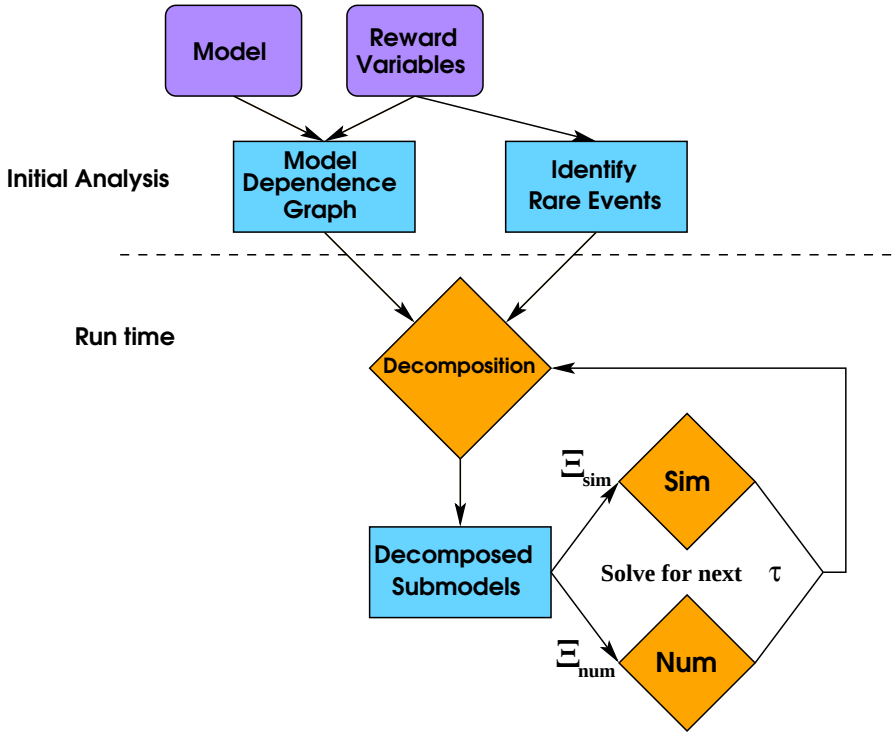


Figure 5.7: Overview of solution Method

$\Xi = \{\xi_0, \xi_1, \ldots, \xi_n\}$ that can be considered independent in the absence of the firing of a rare event. We also discuss how to repartition M using G_M after a rare event has fired, producing a new set of independent submodels, as illustrated in Figure 5.7. When visualizing our simulation as a time-line, as shown in Figure 5.8, we represent the firing of a rare event with the symbol τ. Each time a rare event fires, it results in a reevaluation of our decomposition of M, represented in Figure 5.7 as τ. Given a model and a set of reward variables, we generate an MDG and a set of identified rare events, E_R. Using the MDG, G_M, the set of rare events, E_R, and the current values of all state variables in the model M, we produce

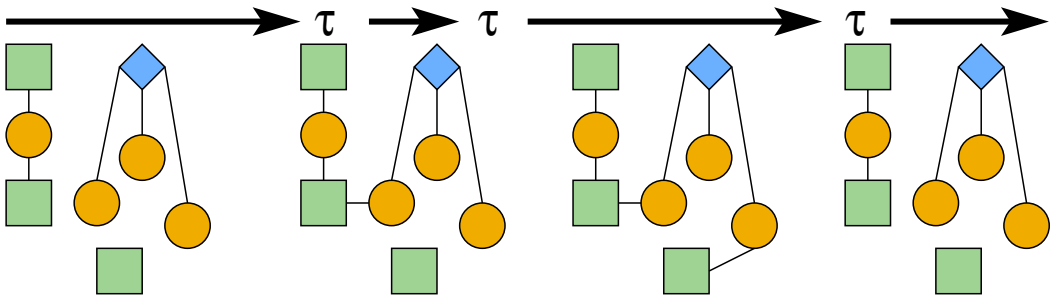


Figure 5.8: Model repartition after each rare event.

a decomposition of the model M into a set of submodels Ξ_R and $\Xi_{!R}$. From Ξ_R, $\Xi_{!R}$, and the subset $\Xi_{E_R} \in \Xi$ of all submodels (in both Ξ_R and $\Xi_{!R}$) that contain events in E_R, we produce two new sets of submodels:

$$\Xi_{\text{Sim}} = \Xi_R \cup (\Xi_{!R} \cap \Xi_{E_R}) \tag{5.7}$$

$$\Xi_{\text{Num}} = \Xi_{!R} \setminus (\Xi_{!R} \cap \Xi_{ER}). \tag{5.8}$$

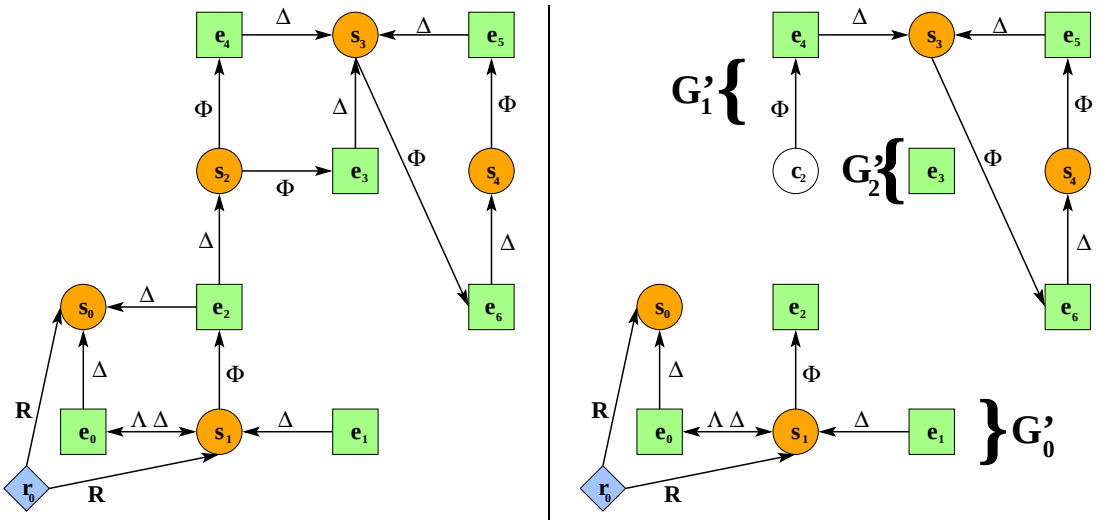
The submodels in the sets Ξ_{Sim} and Ξ_{Num} will then be passed to an appropriate solution method and solved until a rare event fires, at which point we repeat the decomposition step based on the new state of the model.

By decomposing our model in this fashion, we hope to remove from consideration events for which there is no current direct or indirect dependency from our reward variables in the absence of a rare event firing. In order to form this decomposition, however, we must analyze the dependencies in G_M, and from the results of that analysis form a decomposed model dependency graph G'_M that can be used to identify a submodel decomposition of M. We form a decomposed model dependency graph G'_M for M by first removing all Δ-dependencies that involve events in E_R. For every vertex associated with a state variable whose only Δ-dependencies involve events in E_R, we replace those vertices with new vertices from a set V_C, which represent constant state variables whose values are equal to their initial conditions. All vertices that represent events with rates dependent on state variables that are now represented by constant vertices are examined. If such events have transition rate function specifications such that $\Lambda(e, q) = 0$ for all $q \in \mathbb{N}_1 \times \mathbb{N}_2 \times \ldots \times \mathbb{N}_n$ given V_C, or have enabling function specifications such that $\Phi(e, q) = 0$ for all q given V_C, they are removed. All dependencies of removed events are also removed. The process is repeated, examining all V_S and V_E iteratively until no new vertices are removed. This process for generating G'_M using G_M and E_R is presented in Algorithm 1.

The graph G'_M that results from the application of Algorithm 1 to G_M and E_R is then used to determine if a valid partition of the model M exists for our technique. If G'_M defines multiple unconnected sub-graphs, $G'_M = \{g'_0 \cup g'_1 \cup \ldots\}$, a valid partition exists. If it does not, our technique is not applicable. The sub-graphs of G'_M correspond to the submodels in

Algorithm 1 Remove rare-event-based dependencies from G'_M.

$G'_M = (V', A', L') \leftarrow G_M$
$P \leftarrow E_R$
while $P \neq \emptyset$ **do**
  Remove all edges in A' containing at least one vertex in P. Do not remove vertices with edges labeled Φ or Λ if the vertex is in E_R.
  $P \leftarrow \emptyset$
  for all $v_i \in V'_S$ **do**
    if $!\exists v_i v_j \in A'$ such that $v_i v_j$ has label Δ **then**
      $V' \leftarrow V' \setminus v_i$
      Create a new constant vertex $v_{c_i} \in V_C$
      $V' \leftarrow V' \cup v_{c_i}$
      Associate a value equal to the initial marking of $s_i \in S$ associated with v_i with v_{c_i}
    end if
  end for
  for all $v_j \in V'_E$ **do**
    if $\exists v_i | v_i v_j \in A'$ labeled Φ such that $v_i \in V_C$ **then**
      if $!\exists q$ consistent with the constant markings associated with vertices in V_C and $\Phi(e_j, q) = 1$ **then**
        $P \leftarrow P \cup v_j$
      end if
    end if
    if $\exists v_i | v_i v_j \in A'$ labeled Λ such that $v_i \in V_C$ **then**
      if $!\exists q$ consistent with the constant markings associated with vertices in V_C and $\Lambda(e_j, q) \neq 0$ **then**
        $P \leftarrow P \cup v_j$
      end if
    end if
  end for
  $V' \leftarrow V' \setminus P$
end while



(a) Composed Model (b) Decomposed Model

Figure 5.9: Example decomposition of a model dependency graph G_M to G'_M.

our partition Ξ. For a given sub-graph, $g'_i = (V'_i, A'_i)$, for each $v'_j \in V'_i$ such that $v'_j \in V_S$, we add the corresponding state variable to ξ_i. For each $v'_j \in V'_i$ such that $v'_j \in V_E$, we add the corresponding event to ξ_i. In addition, for each $\xi_i \in \Xi$ we restrict the definitions of $\Phi(e_j, q), \Lambda(e_j, q)$, and $\Delta(e_j, q)$ to $e_j \in \xi_i$ and $q \in \mathbb{N}_1, \mathbb{N}_2, \ldots, \mathbb{N}_{|S_{\xi_i}|}$ such that $S_{\xi_i} \in \xi_i$.

To further explain our decomposition algorithm, we present an example model dependency graph and its decomposition in Figure 5.9. We assume that the event represented by the vertex e_2 is a rare event in the model. Applying algorithm 1, we begin with a set $P = e_2$, remove those edges labeled Δ that involve e_2, and clear P. We note that the state variable represented by s_2 remains constant in the absence of e_2's Δ edge, and replace it with a constant vertex c_2 with a value equal to its initial conditions. We then find that events e_3 and e_4 have dependencies that are marked by Φ labeled as arcs, indicating an external dependency. Assume that the enabling conditions of e_3 are not met by the constant value of c_2, but the enabling conditions of e_4 are met. We add e_3 to the now-empty P and iterate again, this time removing e_3. We do not remove s_3, despite the Δ-dependency, as s_3 has an additional Δ-dependency on e_4. At this point, P is empty, and we exit the algorithm, yielding $G'_M = G'_0 \cup G'_1$.

5.8.1 Mitigation, Recovery, and Propagation Events

In Section 5.7.2 we mentioned that by using methods discussed in this section, we would be able to find mitigation, recovery and fault propagation events. Through execution of Algorithm 1 on M, with state variables set to represent an initially fault-free model, removing state variables and events in the manner described by Algorithm 1, it is guaranteed that recovery, mitigation and propagation events will be removed. This is due to the fact that the fault associated with those actions have yet to occur. Since recovery, mitigation and propagation actions have a direct correspondence with faults in the system (giving them their rare enabling conditions), in the absence of a fault, their enabling conditions cannot be met by the constant placeholders we use to represent the effects of a fault event's firing. Thus those events will be removed during our decomposition step.

When Algorithm 1 is used, the set of all events added to P is the set of fault events in E_R plus any events that depend on E_R; that represent recovery, mitigation, or fault propagation; and that are added to E_R when our model is being decomposed and solved during simulation. An example of this procedure is illustrated in Appendix A.

5.8.2 Analyzing Reward Variable Dependencies

It is important to note that reward variable dependencies were preserved in G'_M. These dependencies prevent decomposition of otherwise independent sub-graphs by maintaining connectivity based on reward dependence and help us choose solution methods for submodels in Ξ. Given some reward variable θ_i, during a given period of solution, the reward variable may be evaluated for the solution period using only information on the state variables and events contained in the submodel it belongs to in a decomposition of G'_M.

Proposition 2. *In the absence of the firing of a rare event, the reward variable θ_i is independent from a submodel ξ_j if no direct dependence exists in G'_M from θ_i to a vertex in g'_j.*

Proof. If a direct dependence existed between a reward variable θ_i and a state or event in ξ_j then G'_M would have an edge connecting θ_i to a vertex in g'_j, and a path would exist. If

there were an indirect dependency between θ_i and a vertex in g'_j, then a path would exist between a vertex v_k that has a direct dependency with θ_i and a vertex in g'_j. Then v_k would be a vertex in g'_j, and θ_i would have an edge connecting directly to a vertex in g'_j. □

Given G'_M, we divide all submodels in Ξ defined by the independent sub-graphs of G''_M into two sets: those upon which reward variables do and do not depend in the absence of rare events. These sets of submodels are called Ξ_R and $\Xi_{!R}$, respectively.

5.9 Solving the Decomposed Model

A variety of solution techniques can be used to solve a decomposable storage system model once the dependencies have been identified and relevant decomposition events, both failures and repair events, have been found. We present in this section an algorithm for hybrid simulation of decomposed models, and a discussion of complementary solution methods from the literature. Our hybrid simulation algorithm was designed to help us study the dependability characteristics of deduplicated data storage systems. In Chapter 6, we will use these methods to solve a real system and will present the results of our solution.

5.9.1 Hybrid Simulation of Rare-Event Decomposed Systems

Our study of rare-event-based decomposition methods was motivated by a desire to study the dependability characteristics of storage systems that utilize data deduplication, in a fault environment characterized by rare events. In order to estimate the value of reward variables defined for models of these systems, we have employed our decomposition methods and a hybrid simulation algorithm.

When solving our model, we view trajectories of model execution as a time series $\tau_0 \longrightarrow \tau_1 \longrightarrow \tau_2 \longrightarrow \tau_3 \longrightarrow \ldots$ where τ_0 represents our start time, and each subsequent τ_i represents the firing of a rare event. The set Ξ_{Sim} has all submodels that contain either a rare event or a reward dependency. The set Ξ_{Num} has all submodels that have neither rare events nor reward dependencies. From Proposition 2, reward variable solution does not depend on Ξ_{Num}. Thus we need only solve the state occupancy probability for all submodels in Ξ_{Num} at the time

Algorithm 2 Hybrid Simulation of M

Given M, Θ_M, G_M and initial values for all state variables.
while Θ_M not converged **do**
  Generate G'_M and Ξ from G_M.
  Derive Ξ_{Sim} and Ξ_{Num}.
  Simulate Ξ_{Sim} until the next event is in the set E_R.
  Generate $\pi^*_{\xi_i}$ for each submodel Ξ_{Num}.
  Generate a random state for $\xi_i \in \Xi_{\text{Num}}$ treating $\pi^*_{\xi_i}$ as the pmf of the random variable.
  Recompose M. Simulate the next rare event in M.
  Use current state of M as the next initial state.
end while

of the next rare event firing. We do so by making the assumption that the submodels enter steady state between firings of rare events. This assumption seems appropriate for two reasons. The first is the high probability of a long inter-event time between rare events. The second relates to the fact that for the storage systems in which we are interested, systems tend to enter into steady state instantly in the absence of rare events. The scrub process, for example, is always in steady state; the same holds true for many recovery, propagation, or mitigation submodels. Simulation of the model M is performed using Algorithm 2.

The general improvement offered by this algorithm comes from the reduction of events that must be simulated in order to estimate the effect of rare events in the system. Bucklew and Radeke [70] give a general rule of thumb that in order to estimate the impact of an event with probability ρ, we must process approximately $100/\rho$ simulations. Our method seeks to reduce the number of events that must processed for each computed trajectory by eliminating those events that cannot impact Θ_M without the firing of a rare event.

The performance improvement offered by this algorithm varies with the model and with the degree of dependence of the state variables and events in the model. For models whose resulting Ξ do not have the proper structure, our proposed hybrid simulator may provide no improvement. Between firings of a rare event, our method will produce a speed-up proportional to the rate at which we remove events from explicit simulation. Thus given E' as the set of all events $e_i \in \Xi_{\text{Num}} \cup e_j \in M$ such that $e_j \notin \Xi$, our improvement is proportional to $\frac{\sum_{e_i \in E'} \lambda(e_i,\psi_i)}{\sum_{e_i \in E} \lambda(e_i,\psi_i)}, \forall \psi_i \in \Psi$. This improvement is due to the removal of events that, while enabled, cannot change the state of our model in a way that influences our reward

variables. For instance, a write process may still be enabled, but in the absence of a UDE the write process cannot result in the propagation of a UDE to parity. Likewise while it is important to represent the position of the scrub process, it cannot result in mitigation of a fault until a fault is present to mitigate. By removing those events from our simulator, we improve the efficiency of our solution.

5.9.2 Required Assumptions

A key assumption of our solution method is that the storage sub-models reach steady state between rare events. For the storage systems of interest, the initial transient period is not part of the system lifetime during production. We consider the initial state of any storage system to be fault-free and with uniform distribution of files across the storage system itself. The steady state of such a system can be considered this fault-free condition, with the placement of files described by the observed empirical distribution used to model deduplication.

Other sub-models that are likely to be considered independent, such as the model of the scrub process, are characterized by periodic processes that are either unperturbed by rare events in the system, or dormant in the absence of faults. Thus, any system that is not currently composed with another system that contains a failure can be said to be in steady state.

5.9.3 Correctness of Reward Variables

In this section, we show the three types of reward variables defined in Section 5.4. We redefine these reward variables for use with our decomposition algorithm and hybrid solution method, described previously. We demonstrate that the resulting reward variables are equivalent to those defined in Section 5.4.

When solving for instant-of-time variables, we use the same equation given in Section 5.4:

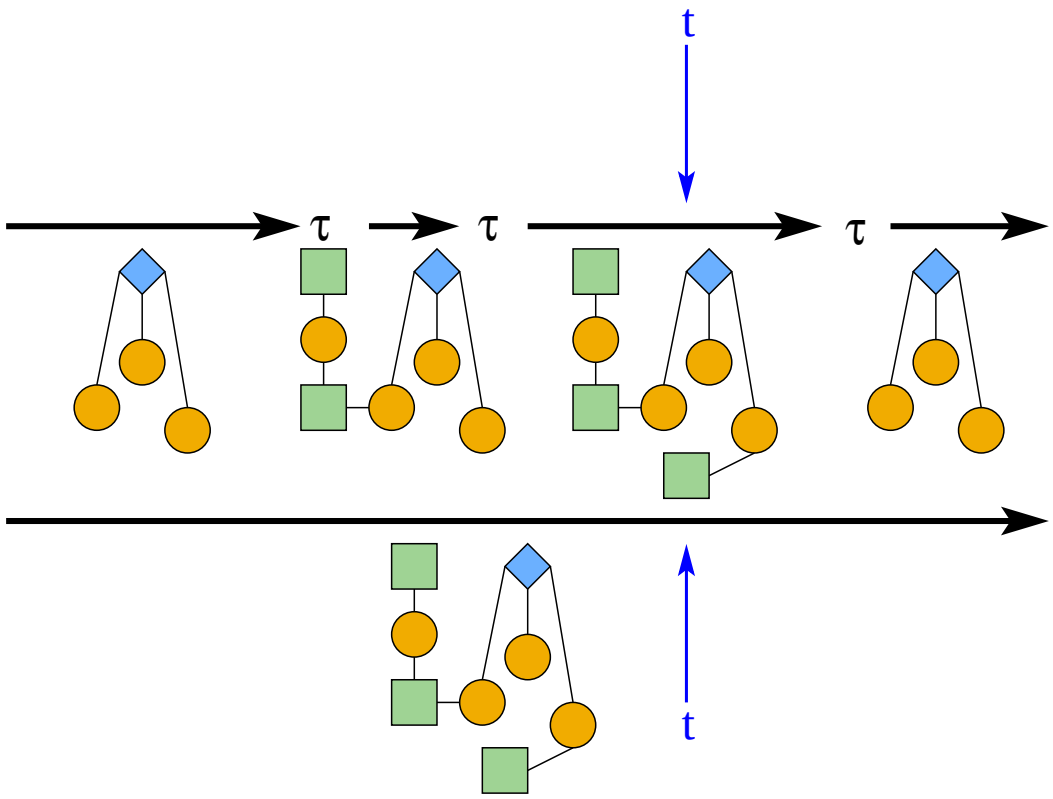


Figure 5.10: Comparison of instant-of-time reward variable solutions.

$$\theta_t = \sum_{\nu \in \mathcal{P}(S,\mathbb{N})} \mathcal{R}(\nu) \cdot I_t^{\nu} + \sum_{e \in A} \mathcal{I}(e) \cdot I_t^{e} \tag{5.9}$$

We evaluate this variable in the same fashion for the submodel that contains the necessary state variables to establish each $\nu \in \mathcal{P}(S, \mathbb{N})$ and each $e \in A$.

Proposition 3. *For a given model M, a decomposed MDG G'_M, and an instant-of-time reward variable θ_t, solving for θ_t using Equation 5.9, and the appropriate submodel decomposition proposed by G'_M yields the same result as the original model M.*

Proof. From Proposition 2 we know that the reward variable θ_t is independent from a submodel ξ_j at time t if no direct dependence exists in G'_M from θ_t to a vertex in g'_j. Thus the solution at time t for θ_t for our decomposed submodels is the same as the solution for M. □

Because of our method of forming an MDG, reward dependencies will exist between a variable and all state variables and events, ensuring that the submodel is decomposed in such a way that the resulting submodel contains everything necessary to evaluate θ_t, as illustrated in Figure 5.10. As shown, at the time we evaluate our reward variable, t, all state variables and rewards on which our reward variable depends at time t are in the same submodel as the reward variable.

To solve for interval-of-time variables using our hybrid solution method, we must provide a new method of computation. Recall from Section 5.4 that an interval-of-time variable $\theta_{[t,t+l]}$ is defined as follows:

$$\theta_{[t,t+l]} = \sum_{\nu \in \mathcal{P}(S,\mathbb{N})} \mathcal{R}(\nu) \cdot J^{\nu}_{[t,t+l]} + \sum_{e \in A} \mathcal{I}(e) \cdot N^{e}_{[t,t+l]} \tag{5.10}$$

We modify the computation of interval-of-time variables to accommodate our solution technique by using multiple random variables for $J^{\nu}_{[t,t+l]}$ and $N^{e}_{[t,t+l]}$ based on the decomposition and re-composition of the underlying model, as dictated by our solution techniques.

As shown in Figure 5.11 we have a set of n model decompositions that form intervals defined by the times $d_0, d_1, \ldots, d_{n-1}$ during the period $[t, t+l]$. For these intervals, we create $n+1$ random variables to replace $J^{\nu}_{[t,t+l]}$ and $N^{e}_{[t,t+l]}$:

$$J^{\nu}_{[t,d_0]}, J^{\nu}_{[d_0,d_1]}, \ldots, J^{\nu}_{[d_{n-1},t+l]}$$

$$N^{e}_{[t,d_0]}, N^{e}_{[d_0,d_1]}, \ldots, N^{e}_{[d_{n-1},t+l]}$$

Each of these random variables is equivalent to those from the previous definition, but over a different interval of time.

The variables are distinguished by $n+1$ separate intervals in the set

$$D = \{[t, d_0], [d_0, d_1], \ldots, [d_{n-1}, t+l]\}$$

Based on those identities, we redefine the method for calculating an interval-of-time variable

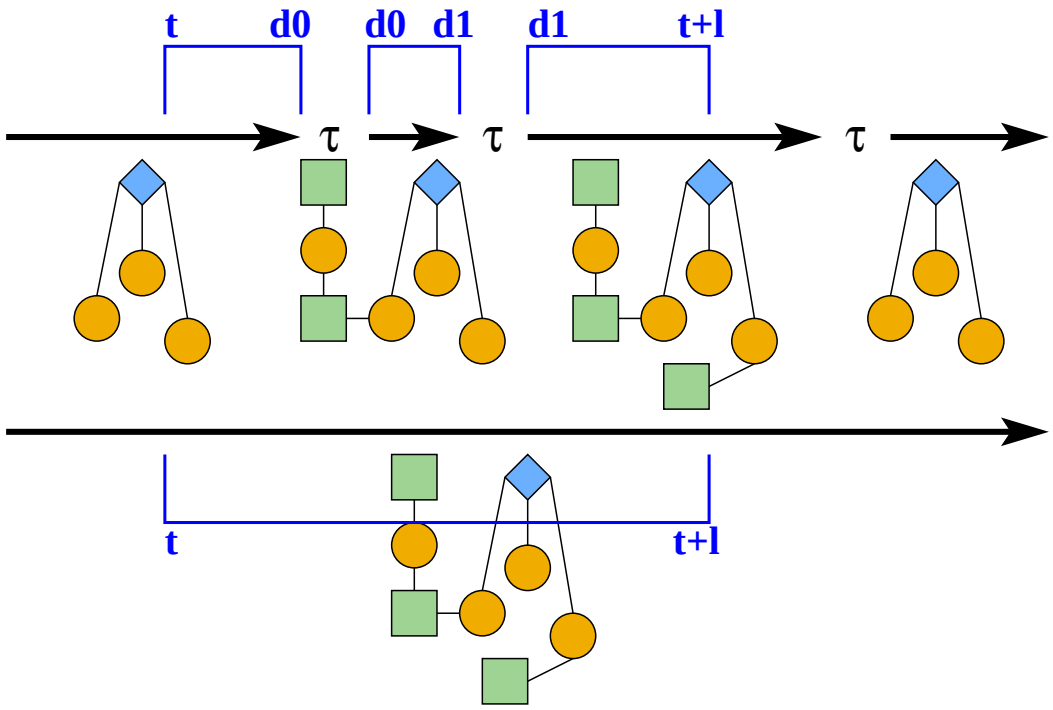


Figure 5.11: Comparison of instant-of-time reward variable solutions

for our solution method as follows:

$$Y_{[t,t+l]} = \sum_{\nu \in \mathcal{P}(S,\mathbb{N})} \sum_{d \in D} \mathcal{R}(\nu) \cdot J_d^{\nu} + \sum_{e \in A} \sum_{d \in D} \mathcal{I}(e) \cdot N_d^{e} \tag{5.11}$$

The differences between that calculation and the one shown in Section 5.4 are illustrated in Figure 5.11. In the original model, we use a single random variable for each rate and impulse reward. Using our methods, however, we need one for each separate interval in D. The two methods are actually equivalent, however, as the sum of the new indicator variables yields the old indicator variables. The reason is that the decomposition algorithm we have presented preserves reward dependencies.

Proposition 4. *For a given model M, a set of decomposed MDGs $\hat{G'_M}$ over the interval $[t, t+l]$, and an interval-of-time reward variable $Y_{[t,t+l]}$, solving for $Y_{[t,t+l]}$ using Equation 5.11 and the appropriate submodel decompositions for each interval in D yields the same result as the original model M.*

Proof. Starting from the definition from Equation 5.11, we prove the equivalence of $Y_{[t,t+l]}$ and $\theta_{[t,t+l]}$ by construction.

$$Y_{[t,t+l]} = \sum_{\nu \in \mathcal{P}(S,\mathbb{N})} \sum_{d \in D} \mathcal{R}(\nu) \cdot J_d^\nu + \sum_{e \in A} \sum_{d \in D} \mathcal{I}(e) \cdot N_d^e \tag{5.12}$$

$$= \sum_{\nu \in \mathcal{P}(S,\mathbb{N})} \mathcal{R}(\nu) \sum_{d \in D} J_d^\nu + \sum_{e \in A} \mathcal{I}(e) \sum_{d \in D} N_d^e \tag{5.13}$$

Given Proposition 2, which states that all state variables and events required for calculating a reward variable are contained within the submodel containing the reward variable itself, we have that:

$$J_{[t,t+l]}^\nu = \sum_{d \in D} J_d^\nu \tag{5.14}$$

$$N_{[t,t+l]}^e = \sum_{d \in D} N_d^e \tag{5.15}$$

The sums of our new indicator variables are the original indicator variables from Section 5.4. Thus,

$$Y_{[t,t+l]} = \sum_{\nu \in \mathcal{P}(S,\mathbb{N})} \mathcal{R}(\nu) J_{[t,t+l]}^\nu + \sum_{e \in A} \mathcal{I}(e) N_{[t,t+l]}^e \tag{5.16}$$

$$= \theta_{[t,t+l]}. \tag{5.17}$$

Interval-of-time reward variables calculated with our method are equivalent to those calculated with typical discrete event simulators.

□

For time-averaged interval-of-time variables, we redefine the variable as

$$W_{[t,t+l]} = \frac{Y_{[t,t+l]}}{l}. \tag{5.18}$$

The above calculation is similar to one given in Section 5.4, but uses the method for calculating interval-of-time variables that takes into account the subdivisions of the interval $[t, t+l]$ given by D.

Proposition 5. *For a given model M, a set of decomposed MDGs $\hat{G'_M}$ over the interval $[t, t+l]$, and a time-averaged interval-of-time reward variable $\theta'_{[t,t+l]}$, solving for $W_{[t,t+l]}$ using Equation 5.18 and the appropriate submodel decompositions for each interval in D yields the same result as the original model M.*

Proof. Given equation 5.17, 5.18 is equivalent to the definition presented in Section 5.4. From Equation 5.17, we know that $Y_{[t,t+l]} = \theta_{[t,t+l]}$. Substituting $\theta_{[t,t+l]}$ for $Y_{[t,t+l]}$ in Equation 5.18 yields Equation 5.5. □

In this section we have detailed a method to identify all dependence relationships in a model, M, using an MDG, G_M. We then detailed how to identify rare faults in the model. Using the set of identified faults, E_R, and the MDG, we showed how to enlarge the set E_R to include mitigation, repair, and propagation actions. We then showed how to use E_R with G_M to form a set of decomposed submodels, Ξ, which we then solved using Algorithm 2. In the next chapter, we will use these methods to calculate reliability metrics for a real system and draw conclusions about the impact of deduplication on the reliability of data.

CHAPTER 6

ANALYZING THE RELIABILITY OF A DEDUPLICATED STORAGE SYSTEM

6.1 Introduction

In Chapters 2, 3, 4, and 5 we have shown how to build models of complex systems using a component-based methodology, provided detailed models of faults which impact modern storage systems, and discussed methods for efficient solution of models based in these methods. In this chapter we apply the material of the previous chapters to a real deduplicated file system in order to understand the impact of deduplication on the reliability of storage systems.

6.2 Motivation

Data deduplication is increasingly being adopted to reduce the data footprint of backup and archival storage, and more recently has become available for near-line primary storage controllers. Scale-out file systems, highly scalable grid-based network-attached storage systems, are increasingly diminishing the silos between primary and archival storage by applying deduplication to unified petabyte-scale data repositories spanning heterogeneous storage hardware. Cloud providers are also actively evaluating deduplication for their heterogeneous commodity storage infrastructures and ever-changing customer workloads.

While the cost of data deduplication in terms of time spent on deduplicating and reconstructing data is reasonably well understood [14, 15], its impact on data reliability is not, especially in large-scale storage systems with heterogeneous hardware. Since traditional deduplication keeps only a single instance of redundant data, it magnifies the negative impact of data loss. Chunk-based deduplication [16, 5] divides a file into multiple chunks,

meaning the loss of one chunk will create many lost chunks in the storage system. Delta encoding [17, 19, 20, 21] deduplicates at the file level, storing the differences among files, and creating the potential for losing multiple files when ever a file is lost.

Administrators and system architects have found that understanding the data reliability of their system under deduplication to be important but extremely difficult [22]. Deduplication itself poses two potential reliability problems. If any one of the devices upon which it depends fails, the file itself is lost. This additional dependence may be counter-balanced, however, depending on the degree of additional storage efficiency. By decreasing the number of disks in the system with deduplication, we decrease the number of expected disk faults as well. Understanding these complex relationships requires understanding the nature of deduplication itself, as well as the complex interactions created by the underlying storage system.

The existing literature relies on heuristics to address the issue of reliability in deduplication systems; the key recommendation is to keep multiple copies of a data chunk instead of storing only a single instance. The creators of Deep Store [5] proposed to determine the level of redundancy for a chunk based on a user-assigned value of importance. It has been suggested by D. Bhagwat et al. [16] that the number of replicas for a chunk should be proportional to its reference count, i.e., the number of files sharing the chunk. A gap exists in the current literature on the topic of quantifying the data reliability of a deduplication system or providing a means to estimate whether a set of reliability requirements can be met in a deduplication system.

A quantitative modeling of reliability in a deduplication system is nontrivial, even without taking into account the petabyte scale of storage systems. First, there are different types of faults in a storage system, including whole disk failures [41], latent sector errors (LSEs) [9, 10], and undetected disk errors [47, 40, 55]. To consider all these faults together, it is necessary to have an understanding of how these faults manifest, and have a representative model that takes into account dependencies and correlations with other, similar, faults as well as the interactions of independent faults in the hardware environment. Second, these faults can propagate due to the sharing of data chunks or chaining of files in a deduplication system. In order to correctly understand the impacts of these faults and their consequences

on the reliability of our storage system, we need to accurately model both the storage system faults and faults due to data deduplication. We call storage system faults and faults due to deduplication *primary* and *secondary faults* respectively, and discuss them in more detail in Section 6.5. Third, it is important to note that many of the faults we wish to consider are rare compared to other events in the system, such as disk scrubbing, disk rebuilds, and I/O. Calculating the impact from rare events in a system can be computationally expensive, motivating us to find efficient ways of measuring their effect on the reliability metrics of interest.

The complexity of this problem arises from two different causes. The first is the state-space explosion problem which can make numerical solution difficult. As our model grows increasingly complex the state space grows rapidly. The simplified deduplication system studied in [71] quickly grows to unmanageable size, having 10^{23} states with only 10 storage subsystems, and 10^{222} states with 100 storage subsystems, exceeding the capabilities of numerical solvers. A second issue comes from the stiffness that results from rare events. For numerical solutions stiffness introduces numerical instability, making solution impractical. When simulating stiffness increases the number of events we must process, causing a resulting increase in simulation complexity. These factors, and a desire to precisely understand the complex relationships present a need to use more sophisticated methods of analysis to fully understand the implications of deduplication.

6.2.1 Our Contributions

In this chapter, we utilize the hybrid simulation approach detailed in Chapter 5 to quantitatively analyze the reliability of a modeled deduplication system with heterogeneous data on heterogeneous storage hardware, in the presence of primary faults and their secondary effects due to deduplication. The analysis is based on three key dimensions that our model takes into account:

- The fault tolerance characteristics of the underlying storage hardware.
- The statistical deduplication characteristics (e.g., reference count distribution) of a set

of deduplicated data.

- The number of replicas of a given data chunk that are stored.

To validate our modeling approach, we studied data from an enterprise backup storage system containing 7 terabytes of deduplicated data comprising over 2.7 million unique file names and over 193 million references to 2.87 million unique deduplicated data chunks. We analyzed the statistical properties of the real data, including the deduplication relationship implied by references from each file to deduplicated chunks. To a user, different data sets usually have different levels of importance and different reliability constraints. Treating all files the same way is not the right strategy. Therefore, we break our analysis out into twelve separate categories that are based on the file types and applications, characterizing each category separately.

We derived a model of the data that has the same statistical characteristics as the original data set, and evaluated the reliability impact of data deduplication on a variety of different storage hardware with different reliability characteristics, different data categories with different deduplication characteristics, and different numbers of replicas for deduplicated chunks.

The primary goal of this chapter is to improve our understanding of deduplicated storage systems. In doing so we also detail a modeling framework to evaluate the reliability of other deduplication systems with varying hardware reliability characteristics, varying deduplication characteristics, and a varying number of copies of data chunks. We built our models based on a given enterprise system using fault models described in the literature, but our methods can be applied to other systems. In the remaining sections of this chapter, we will not only show the results for this system we studied but detail the methods we used to model this system given appropriate data, so that the same methods can easily be applied to other systems.

We also provide a method for analyzing the effect of multi-copy deduplication on the reliability of our studied storage system. We use our methods to determine the number of copies of various data chunks or files needed to meet a specific reliability requirement while minimizing storage utilization. For our system, we utilized category information along with

reference counts to determine the importance of a deduplicated chunk. We believe that this is preferable to methods based only on the reference count of the deduplicated chunk, as in [5, 16].

In this chapter, we model offline storage-side deduplication implemented through variable chunk hashing. Our models can easily be extended to support other deduplication algorithms, including fixed-size chunk hashing, whole file hashing, and delta encoding. Other deduplication architectures can also be easily modeled, such as online storage-side and online client-side architectures. Those extensions would involve computation of new empirical distributions for the underlying deduplication system.

6.2.2 Related Work

While other studies have approached the question of reliability in data deduplication, they tend to assess impact through an assumption that the number of files referencing a deduplicated chunk is directly proportional to the importance of a chunk [5, 16]. While this may be an appropriate assumption when studying data whose types are largely homogeneous [5, 16], we believe this provides a limited picture of how deduplication affects fault tolerance on real systems storing a large amount of heterogeneous data. Moreover, these studies [5, 16] do not provide a way to quantify the data reliability of a deduplication system.

Our study differs in that it quantitatively analyzes the reliability of a deduplication system with heterogeneous data and heterogeneous storage hardware. We use our methodology to provide insight to help meet design goals for reliability while maintaining an improved storage efficiency over a non-deduplicated system. Our quantitative analysis is performed utilizing discrete event simulation and by exploiting identified near-independent relationships in the underlying model to more efficiently solve a complex model in the presence of rare events as described in [71].

6.3 Overview of Our Reliability Analysis Methodology

The modeling methodology is composed of two main components: a deduplicated file system model and a hardware system reliability model. These models are solved using a the methods discussed in Chapter 5. Given the data to be stored in a deduplicated storage system and the deduplication algorithm used on the system, including parameters such as chunk size and similarity measures, we built a model of deduplication in our storage system that represents the resulting deduplication process. Data in our system are categorized into different classes based on either known application file extensions or user-specified criteria. Our deduplication model summarizes the relationships implied by deduplication for each of these classes. Section 6.4 describes the process by which we characterize data, along with its application on a real data set.

The reliability model of our hardware makes use of system level configuration information to build an on-line mathematical representation of our hardware environment. Specifically, parameters that have more impact on reliability are considered, such as the type of disks used in our system (nearline or enterprise), RAID type (1, 5, and 6) or erasure codes, the size of a stripe on our arrays of disks, and the number and configuration of disks in a reliability group. We model three types of storage-level faults explicitly, including whole disk failure, latent sector errors, and undetected disk errors. Secondary faults due to deduplication are deduced via our model of the deduplicated file system. An in-depth discussion of these topics is provided in Section 6.5.

Given the model of deduplication, the hardware reliability model, and the parameters of the target system (such as the replication factor and data distribution), our discrete event simulator provides per-category estimates of expected reliability for our hardware, deduplicated file system, and parameter sets. Section 6.6 summarizes the results of one such estimation on several hardware systems using a real-world data set and various deduplication parameters.

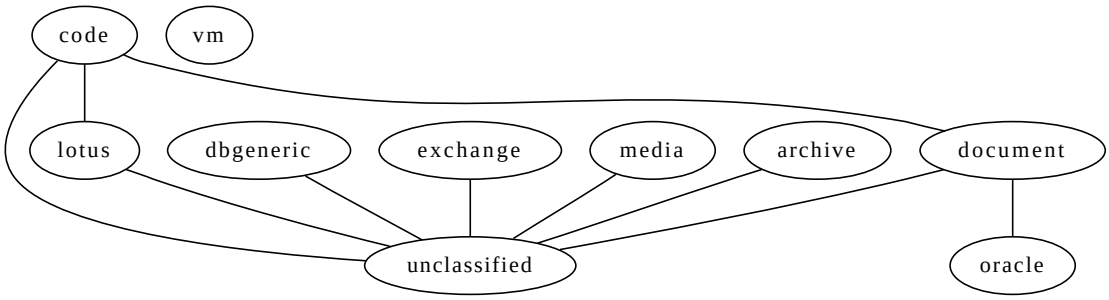


Figure 6.1: Summary of categories that contain files that share deduplicated chunks with other categories.

6.4 Deduplicated File System Description and Model

In order to evaluate the effects of data deduplication on fault tolerance in a real system, we examined the deduplicated data stored in an enterprise backup/archive storage system that utilizes variable-chunk hashing [72, 73]. Our example is a client/server system that provides backup and archive solutions for a multi-vendor computer environment that can consist of file servers, workstations, application servers, and other similar systems. Server-side data deduplication is used for space management in this system.

We present a general model of the relationships implied by data deduplication, and their consequences for fault tolerance, based on our analysis of the real system. We also present refinements to the model necessary to model variable-chunk hashing, as used by the system represented in our data. In order to demonstrate the flexibility of our model of deduplication, we will also show how to adapt it for delta encoding.

6.4.1 Data Analysis

The data stored in the system consist of backups from about 150 workstations (the most common operating system family for workstations in the backup group was Windows, though MacOS and Linux were also represented), as well as backups of several IBM DB2, Oracle, and Microsoft SQL database servers and several mail servers, including IBM Lotus Domino and Microsoft Exchange. The data-set has approximately 7TB of deduplicated data. Without deduplication, the system would require over 16TB to store all the data. The deduplicated data have a total of 193,205,876 separate references to 2,870,681 unique data chunks.

In order to better understand the deduplication relationships implied by our data, we placed all files on the system into eleven categories based on their file extensions. A total of 2,735,894 unique file names were processed, featuring 55,688 unique file extensions. Of these file extensions, only 14,910 appeared more than once, and only 1,520 appeared five times or more. We identified four major categories and eleven subcategories based on file extensions.

- *Databases*: We specified four categories for files associated with database applications **db2**, **Oracle**, **SQL** and **DBGeneric**. We use DBGeneric for those files we know to be used by a database, but for which the specific database is unknown.

- *Mail*: We identified two categories of files associated with mail applications: **Lotus Domino** and **Exchange**.

- *User Data*: We specified four categories for user data files: **Archives**, **Documents**, **Media** and **Code**.

- *VM*: We grouped all virtual machine application data into a single category, **VM**.

We call our twelfth category **Unclassified** and use it to hold system files we assume to be re-creatable from installation media or other sources that make recovery of the data possible, files whose extensions do not match expected for our previous eleven categories, and those files with no file extensions.

We do not suggest that these categories are the best or only ways to partition a deduplicated file system. In fact, we assert that the proper way to partition a file system into categories is context-sensitive and user-specific, based on legal and contractual obligations as well as administrator goals. Categories should reflect groups of important files or applications. To understand the relationships that these categories of files shared through deduplication, we constructed a graph with a set of nodes N_I with an element for every deduplicated chunk in our deduplicated system, and a second set N_C with a node for each category of file defined. When we encountered a reference in the data from a category to some deduplicated chunk, we added the edge connecting the nodes, allowing duplicate edges. The weight of an edge is equal to the number of duplicate edges and defines the number of references to a given deduplicated chunk.

		References per Chunk	
	Unique Chunks	**90th Quantile**	**Maximum**
Archive	50,240	24	174,720
Code	895,615	2	105,404
Document	574,222	2	16,128
Exchange	9,288	4	42,442
Lotus	9,790	14	60,216
Media	148,887	4	3,384
MSSQL	16,089	32	280,044
Oracle	30,460	4	21,476
db2	30,810	6	5,194
DBGeneric	20,456	6	77,120
VM	9,328	2	308,934
Unclassified	1,075,851	8	251,542

Table 6.1: Summary of the data obtained from analysis of deduplicated chunks.

Using this graph, we identified 351 chunks with references from exactly two categories, and two with references from exactly three categories. The remaining 2,870,328 chunks had references from only one category. Figure 6.1 shows the paths between nodes in N_C that pass through exactly one node in N_I and no nodes in N_C. It seems likely that the frequent connections between the unclassified node and other nodes in N_C, represented in Figure 6.1, are indicative of a failure to properly classify files by their extensions, or files with misleading extensions. In such cases, it seems safest to treat unclassified files that reference chunks that share an edge with another category C_k as if they are from category C_k. For those nodes shared between two categories C_i and C_j, where neither is the unclassified category, we consider the node a legitimate cross category deduplication. For our analysis we treat a deduplicated chunk as categorized with the highest level of importance of any referring file or category.

The distribution of references to chunks varied based on which categories were connected in our graph. Table 6.1 summarizes some of this information by showing the total number of unique deduplicated chunks with at least one reference to each of the categories, the 90th quantile for references per chunk for those chunks with at least one reference to a given category, and the maximum number of references per chunk for those chunks with at least one reference to a given category.

6.4.2 Using Category Information to Define Importance

Determining the importance of a file on a deduplicated storage system is a difficult proposition, and is likely to be dependent on the needs and legal requirements of the organization operating the storage system. While one could take the approach of keeping additional copies of all files that have a certain reference count, we suggest that this is a poor measure of importance, for two primary reasons. First, it assumes that chunks with few references are less important. While it is true that the loss of a chunk with fewer references will cause fewer secondary failures due to deduplication, deduplicated chunks fail at the same rate regardless of their reference count, and those chunks with fewer references may be referenced by critical data whose reliability is just as important as that of files that share a chunk with many other files. Second, using reference count as a measure of importance can result in a loss of storage efficiency to increase the reliability of files that are either easily re-creatable system files, or files unimportant to the organization.

6.4.3 Model of a Deduplicated File System

In order to construct a model of our deduplicated file system for use with our methods from Chapter 5, we must first develop an empirical understanding of deduplication in a real file system. We view deduplication on this file system as a dependence relationship and construct a graph, whose nodes represent files and deduplicated chunks in our file system, to model this dependence relationship. Each deduplicated chunk in our file system is represented by a node $n_i \in N_I$. Files themselves are represented by the set $N_F = \{N_{F,C_1}, N_{F,C_2}, N_{F,C_3}, \ldots, N_{F,C_{12}}\}$, where each subset N_{F,C_k} contains a node n_{j,C_k} for each file f_j that is a member of category C_k. Deduplication relationships are again represented by the set of edges E such that if a chunk n_i is referenced by a file f_j in category C_k, an edge $n_{j,C_k}n_i \in E$.

We suggest using the data summarized in Table 6.1 as an empirical estimate of the probability density function (pdf) for a random variable representing the number of references for a chunk in the given category c. Using this pdf, $f_c(x)$, we define an inverse distribution function (idf) $F_c^* : (0,1) \to \mathcal{X}$, defined for all $u \in (0,1)$ as follows:

$$F_{\mathrm{c}}(x) = Pr(X \leq x) = \sum_{t \leq x} f_{\mathrm{c}}(t) \quad (6.1)$$

$$F^{*}_{\mathrm{c}}(u) = \min_{x}\{x : u < F_{\mathrm{c}}(x)\} \quad (6.2)$$

Using $F^{*}_{c}(u)$ and a uniform random variate $\mathcal{U}$, we can generate realizations of the random variable described by $f_c(x)$, allowing us to use our observations summarized in Table 6.1 to synthetically create a deduplication system with the same statistical properties as our example system. The number of edges connecting to any node $n_i \in N_I$ is defined by first determining the primary category of files that refer to the chunk, and by using Equation 6.2 to generate a realization of the random variable described by $f_{C_k}(x)$.

While our study concerns only data deduplication that uses variable-chunk hashing, it is a simple matter to adapt this model for delta encoding or whole file hashing. In those cases, we simply remove the set N_I and define edges between elements of N_F directly. Then the edges in the E must be represented as directed edges of the form $n_{f_a}\vec{n}_{f_b}$, indicating that the file n_{f_a} depends on n_{f_b}. Directed edges are not required for our variable-chunk hashing representation, as it is implied that the relationship is a dependence of nodes in N_F on nodes in N_I.

6.5 Hardware Reliability Models

When modeling hardware, we utilize models of traditional disk failures, LSEs and UDEs. Traditional disk failures are assumed to be non-transient and unrepairable without drive replacement. LSEs can be either transient or permanent [9]. It is important to note that even in the case of a transient LSE, previous study of LSEs has indicated that data stored in the sector are irrevocably lost, even when the sector can later be read or written to properly [9]. In our system model, we consider LSEs to be correctable either when the disk is subsequently rebuilt due to a traditional disk failure, or upon performance of a scrub of the appropriate disk. UDEs represent silent data corruption on the disk, which is undetectable by normal means [13, 47, 40]. UWEs are persistent errors that are only detectable during

a read operation subsequent to the faulty write. We consider UDEs to be correctable when the disk is rebuilt because of a traditional disk failure, upon performance of a scrub of the appropriate disk, or when the error is overwritten before being read, although this type of mitigation produces parity pollution [40].

6.5.1 Disk Model

In order to understand the effect of faults in an example system, we utilize a formal model of disks in our underlying storage system. Each disk in our system is modeled as a set of blocks. The state of a block is modeled using the variable `block_state`. This variable indicates whether the block is in a non-faulty state, or is faulty due to an LSE, UDE, or total disk failure; those possibilities are represented as $0, 1, 2$, or 3, respectively. Each block model contains events which represent faults, fault propagation, fault mitigation, and repair. A full representation for a given block is shown in Figure 6.2. The events and state variables in the model are described in detail in Appendix A. A disk is the collection of all block models on a disk that share a single disk failure event.

6.5.2 Fault Interactions and Data Loss

It is important to note that the occurrence of a fault within our system does not guarantee that data loss has occurred. In many cases, the underlying storage system will utilize some form of fault tolerance, such as RAID. For that reason it is important to separate the modeling of faults and errors in our model. For the purposes of our model we consider faults to include traditional disk failure, LSEs and UDEs, and errors to include data loss which cannot be recovered, and serving corrupted data silently to the user. In general, for a fault to manifest as a data loss error, we must experience a series of faults within a single RAID unit. How these faults manifest as errors depends on the ordering of faults and repair actions in a time line of system events, as shown in Figure 6.3. In the case of RAID 5, a single failure can usually be tolerated before a data loss event occurs. For RAID 6, two failures can be tolerated before data loss. UDEs cause a different kind of error, which is largely orthogonal

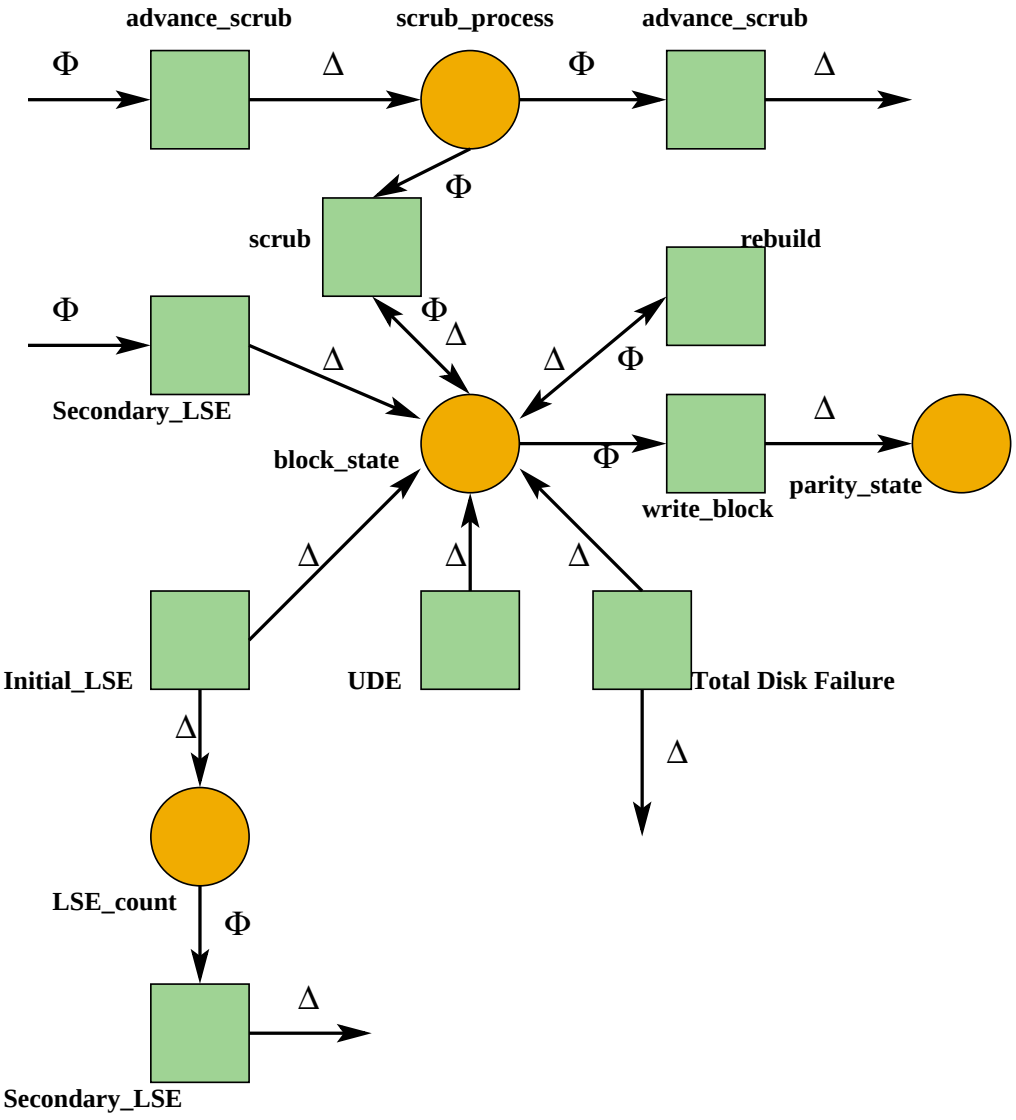
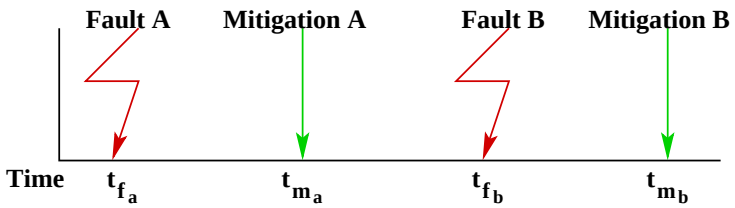
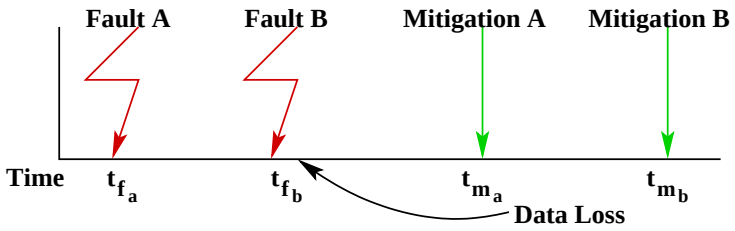


Figure 6.2: Block model diagram



(a) In the case when subsequent fault events arrive to the system after a mitigation event for all previous faults has been processed, there is no potential interaction.



(b) Subsequent faults that arrive to the system before mitigation events for previous faults have the potential to result in data loss.

Figure 6.3: Example fault interactions.

to RAID, by silently corrupting data which can then be served to the user.

In order to determine if a combination of primary faults has led to data loss and potentially a number of secondary faults, we examine the timing of events in a manner similar to the window of vulnerability method described by [74]. Given a storage system that can tolerate n faults before data loss occurs, we will see faults manifest as data loss only when the joint effect of n faults occurs on overlapping portions of disks in the same reliability group before mitigation. To evaluate that, we utilize the representations of the faults on the disk as defined in Section 6.5.1.

Faults in the form of traditional disk failures can result in data loss if their arrival times t_{f1}, t_{f2} are such that for the time at which the initial fault is mitigated $(m_{f1} > t_{f2}) \wedge (m_{f2} > t_{f2})$. In such a case, the entire drive is lost to the failure.

Traditional disk failures can also result in data loss when combined with a subsequent LSE on a read operation. Again, given arrival times of the failure events t_{f1}, t_{f2} and a mitigation time for the first fault m_{f1} $(m_{f1} > t_{f2}) \wedge (m_{f2} > t_{f2})$, an LSE on another disk in the RAID group that corrupts data on the disk before mitigation will result in the rebuilding of an unrecoverable sector on the disk.

UDEs form a special case of fault. While they can be detected by a scrub operation, repair

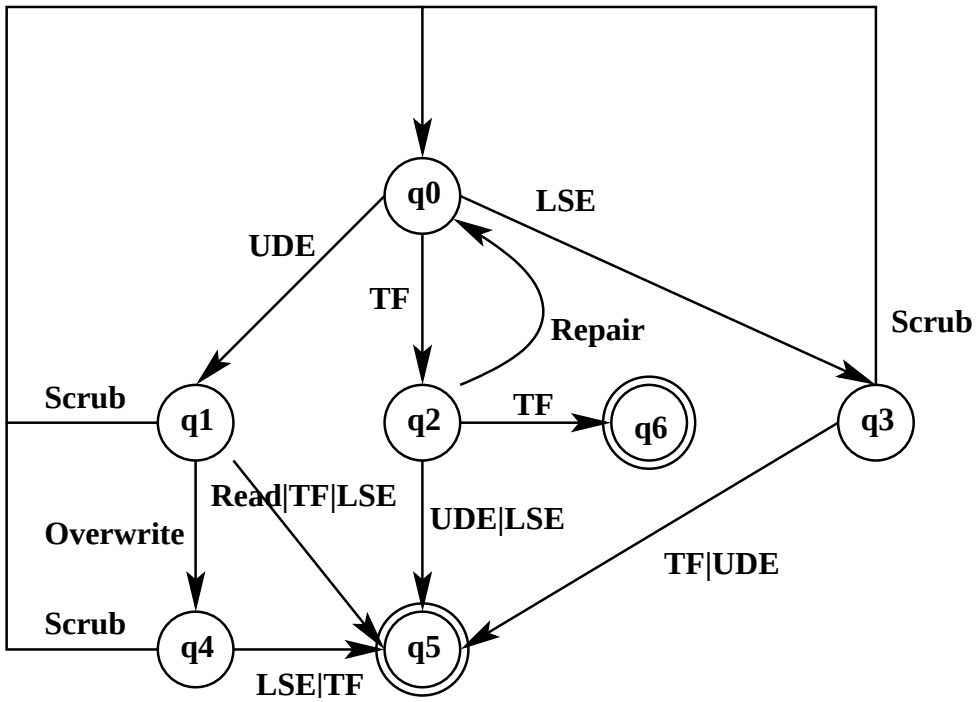


Figure 6.4: DFA representing the combination of faults which lead to data loss on a stripe from UDEs, LSEs, and traditional failures under RAID1, or RAID5.

is not possible. Scrubbing a disk tells us that an error is present in the stripe, but not where the error is. An error in the parity or on any of the data drives creates an identical situation for the scrub.

In order to characterize the interactions of faults in our model, we maintain a state-based model of portions of the physical disk, as detailed by Section 6.5.1. Given a set of disks that are grouped into an interdependent array (such as the set of disks in a RAID5 configuration, or a pair of disks that are mirrored), each stripe in the array maintains its state using a state machine appropriate to the number of tolerated faults the configuration can sustain without data loss, such as shown in the example in Figure 6.4.

Each stripe is represented by a tuple $(Q, \Sigma, \delta, q_0, F)$. The set of states Q can be partitioned into three different subsets; $Q_{good} = \{q_0\}$, the fault-free non-degraded state and start state; $Q_{degraded} = \{q_1, q_2, q_3, q_4\}$, states in which the stripe has suffered a fault but no data loss; and $Q_{fail} = F = \{q_5, q_6\}$, which represent the states that indicate that data have been lost. When the simulator processes an event for a given stripe, it forwards information on the processed event to the state machine in the form of the DFAs input alphabet, $\Sigma =$

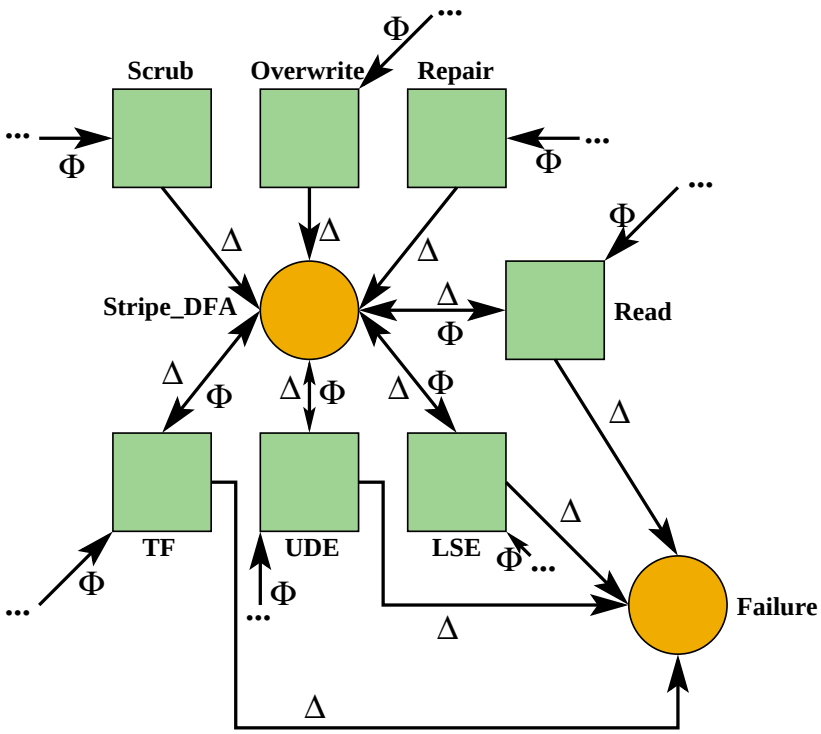


Figure 6.5: Stripe model diagram

$\{TF, LSE, UDE, Write, Read, Scrub, Repair\}$. Each of those symbols represents a fault, a mitigation, or an action that causes a UDE to serve corrupt data undetectably. The DFA transitions on these symbols based on the transition relation defined by $\delta : Q \times \Sigma \to Q$.

A DFA of that form is stored for every stripe in a disk group. The state of the DFA is stored in a state variable called `Stripe_DFA` (as shown in Figure 6.5), which takes on values $\{0, 1, 2, 3, 4, 5, 6\}$ to represent the states in Q for our DFA. Each of the symbols in Σ are events in this stripe model whose enabling conditions are the presence of these faults in a block in the given stripe on the system. Another state variable, `Failure`, is set to 1 when `Stripe_DFA` has a value in $\{5, 6\}$. The DFAs maintained by stripes within our modeled system are generated automatically using knowledge of potential fault interactions and parameters that define the size of the disk array s_{array} and the number of disk faults $n_{\text{tolerated faults}}$ that the array can tolerate without data loss, as defined by the array's RAID level [12].

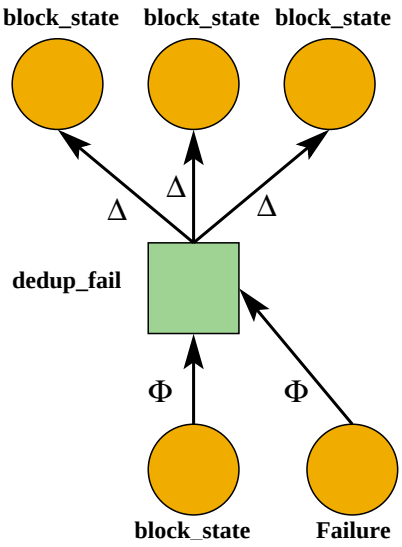


Figure 6.6: Example MDG representing deduplication relationships.

6.5.3 Deduplication Model

When modeling the impact of errors in our system, we utilize a model of deduplication based on the empirical data and analysis from Section 6.4. If a failure of a stripe causes a deduplicated block to be lost, additional blocks will also be lost if the lost block represented a stored deduplicated instance. These secondary faults would not have occurred, were it not for the deduplication strategy used by the storage system. In the case of whole file hashing, delta encoding, or other deduplication methods that allow for a chain of references, the loss of a block implies not only the loss of all blocks that reference the data, but also, recursively, the loss of all blocks associated with files dependent on the reference itself.

We encode those relationships in the model when it is generated as dependence relationships and correlated failures that occur when an underlying block has failed, as shown in the example in Figure 6.6. In the figure, when the `block_state` state variable shown at the bottom of the diagram has failed, and the `Failure` state variable indicates the failure of the stripe an additional failure of deduplicated references occurs because of the loss of a required instance. Multi-copy deduplication modifies the underlying model as shown in Figure 6.7. In that scenario, each deduplicated instance is stored in two blocks, so the failure of both the two blocks and their associated stripes must occur before the enabling conditions of the deduplicated failure are met.

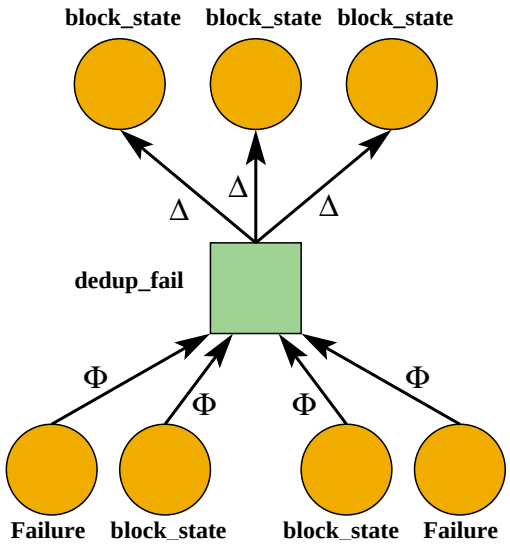


Figure 6.7: Example MDG representing deduplication relationships with 2-copy deduplication.

These relationships are encoded in the model of a storage system by randomly assigning each block a category, then based on the category a status as either undeduplicated data, deduplicated instance, or deduplicated reference as appropriate based on the empirical distributions we calculated from the data. The appropriate events and dependencies are then encoded in the model.

6.6 Discrete Event Simulation Results

In order the understand the impact of data deduplication on fault tolerance we simulated systems with data sets of 7TB (based on the system described in Section 6.4) and 1PB before deduplication. Both systems are assumed to have a deduplication ratio of 0.5. We modeled the systems with reliability provided by various RAID and erasure codes, including RAID1 (mirroring), RAID5 in $7 + p$ and $8 + p$ configurations, RAID6 in an $8 + 2p$ configuration, and erasure codes in an $8 + 3p$ configuration. For each system we calculated two reliability measures: the rate at which all copies of a deduplicated chunk were lost, and the rate at which undetected corrupt data was served to applications.

We make the assumption that our modeled system features a workload of 100 io/s for

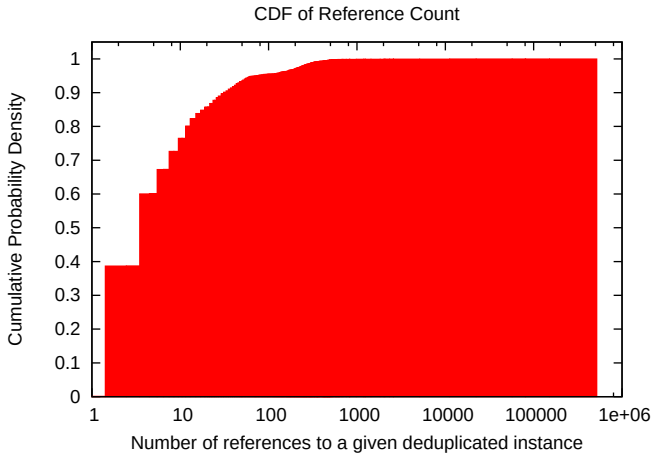


Figure 6.8: Cumulative probability density function of the number of references to each deduplicated instance for the SQL category.

	7 TB			
	RAID1		RAID5	
	1 copy	2 copy	1 copy	2 copy
Archive	1.2e+02 ± 6.2e+01	8.6e-10 ± 2.2e-10	1.7e+03 ± 7.1e+02	1.8e-07 ± 2.9e-0
Code	2.9e+03 ± 6.4e+02	2.8e-08 ± 1.3e-09	6.8e+04 ± 3.0e+04	1.5e-05 ± 2.9e-06
db2	3.3e+03 ± 1.6e+03	5.4e-08 ± 1.3e-08	4.1e+04 ± 2.2e+04	8.6e-06 ± 2.5e-06
DBGeneric	6.5e+01 ± 8.5e+01	1.3e-09 ± 2.3e-09	1.9e+02 ± 1.3e+02	1.2e-08 ± 5.5e-09
Document	6.6e+01 ± 1.1e+02	1.3e-09 ± 3.3e-09	2.2e+02 ± 1.1e+02	1.4e-08 ± 3.5e-09
Exchange	4.0e+02 ± 1.5e+02	3.2e-09 ± 4.1e-10	5.6e+03 ± 1.3e+03	6.1e-07 ± 3.3e-08
Lotus	2.4e+01 ± 2.1e+01	1.1e-10 ± 7.6e-11	4.0e+03 ± 5.2e+03	2.8e-06 ± 5.0e-06
Media	1.3e+03 ± 2.6e+03	1.7e-07 ± 6.4e-07	1.3e+03 ± 8.7e+02	1.7e-07 ± 7.2e-08
Oracle	7.2e+01 ± 4.1e+01	7.4e-10 ± 2.4e-10	4.8e+02 ± 2.8e+02	3.3e-08 ± 1.1e-08
SQL	7.2e+01 ± 4.1e+01	4.9e-10 ± 1.6e-10	1.2e+03 ± 6.8e+02	1.3e-07 ± 4.4e-08
Unclassified	4.9e+03 ± 1.5e+03	6.5e-08 ± 6.3e-09	1.6e+05 ± 1.8e+05	7.0e-05 ± 8.4e-05
VM	2.6e+01 ± 2.7e+01	2.1e-10 ± 2.3e-10	2.7e+02 ± 1.1e+02	2.3e-08 ± 4.0e-09

Table 6.2: Estimated rate of file loss per year, for the 7TB system using RAID1 and RAID5, and a single copy of each deduplicated chunk.

	1 PB			
	RAID1		RAID5	
	1 copy	2 copy	1 copy	2 copy
Archive	8.2e+03 ± 4.1e+03	5.7e-08 ± 1.5e-08	1.2e+05 ± 4.7e+04	1.2e-05 ± 1.9e-06
Code	2.0e+05 ± 4.3e+04	1.9e-06 ± 8.8e-08	4.6e+06 ± 2.0e+06	1.0e-03 ± 1.9e-04
db2	2.2e+05 ± 1.1e+05	3.6e-06 ± 8.9e-07	2.8e+06 ± 1.5e+06	5.7e-04 ± 1.7e-04
DBGeneric	4.3e+03 ± 5.7e+03	8.7e-08 ± 1.5e-07	1.3e+04 ± 8.9e+03	7.9e-07 ± 3.7e-07
Document	4.4e+03 ± 7.0e+03	8.5e-08 ± 2.2e-07	1.4e+04 ± 7.3e+03	9.1e-07 ± 2.3e-07
Exchange	2.7e+04 ± 9.7e+03	2.1e-07 ± 2.8e-08	3.7e+05 ± 8.7e+04	4.1e-05 ± 2.2e-06
Lotus	1.6e+03 ± 1.4e+03	7.0e-09 ± 5.1e-09	2.6e+05 ± 3.5e+05	1.9e-04 ± 3.3e-04
Media	8.8e+04 ± 1.7e+05	1.1e-05 ± 4.2e-05	8.9e+04 ± 5.8e+04	1.1e-05 ± 4.8e-06
Oracle	4.8e+03 ± 2.7e+03	4.9e-08 ± 1.6e-08	3.2e+04 ± 1.9e+04	2.2e-06 ± 7.5e-07
SQL	4.8e+03 ± 2.7e+03	3.3e-08 ± 1.1e-08	8.0e+04 ± 4.6e+04	8.9e-06 ± 2.9e-06
Unclassified	3.3e+05 ± 1.0e+05	4.4e-06 ± 4.2e-07	1.1e+07 ± 1.2e+07	4.7e-03 ± 5.6e-03
VM	1.7e+03 ± 1.8e+03	1.4e-08 ± 1.5e-08	1.8e+04 ± 7.6e+03	1.5e-06 ± 2.7e-07

Table 6.3: Estimated rate of file loss per year, for the 1PB system using RAID1 and RAID5, and a single copy of each deduplicated chunk.

each data disk in the system with reads making up 95% of the workload. Disks themselves are assumed to be 750GB with 128k strips, with read and write requests simulated only for those portions of a disk containing data. We assume data is distributed uniformly across all disks, and that reads and writes are likewise uniformly distributed. We derive the rate of traditional disk failures from [41], and latent sector errors from [10] using the parameters given for system $A-1$ in the paper. We derive rates for UWEs are the same as described in [55] for enterprise drives.

Reliability is effected by deduplication in two ways: the *incidence* and *impact* of faults. We found in our simulations, that the incidence of faults is reduced by a factor equal to the deduplication ratio, due to the reduction in the number of disks required to store the same data set. The impact, however, of each fault was increased for all categories of data by a factor larger than the reduction provided by deduplication, resulting in a net decrease in fault tolerance for all categories of files. The impact of a fault during simulation was calculated using empirical cumulative probability distributions calculated from the data set. First the file or files suffering an error were assigned a category randomly, next based on the category, we randomly determined whether the segment of the file or files lost contained a deduplicated chunk based again on empirical distributions from our data. Finally, if the file lost was a deduplicated chunk we generated a random number of references to the chunk, and assigned them randomly generated locations in the storage system so that in the case of errors spanning multiple files, a file and adjacent reference were not double counted when both were lost to the initial error. An example CDF for the number of references to a deduplicated for files in the SQL category is shown in Figure 6.8.

The rate of permanent data loss due to unrecoverable faults is shown in Tables 6.2 and 6.3 for RAID1 and RAID5. The rate of loss increases for all categories when only a single instance is kept per deduplicated chunk. For configurations with higher fault tolerance (RAID6 and $8+3p$ erasure codes), we saw no significant decrease in fault tolerance during the expected lifespan of a typical storage system. Any increase in the impact of data deduplication on the unrecoverable loss of system data is masked by the low incidence of unrecoverable data loss during the expected system lifespan in the systems we studied.

The dramatic improvements witnessed when two copies are kept of each deduplicated

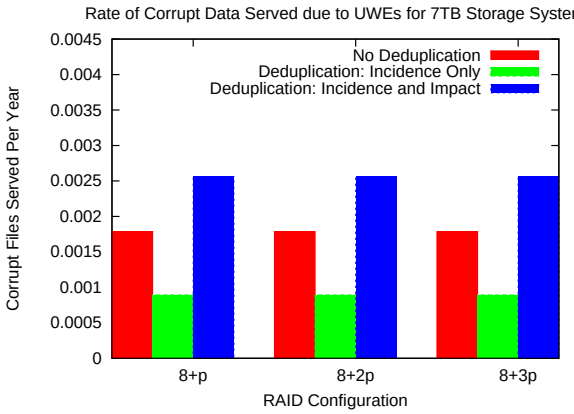
chunk are to do to the circumstances which must occur in order to permanently lose the data stored in the chunk. In addition to the requisite correlated faults shown in Figure 6.3, the same situation must occur on the independent storage unit which holds the other copy of the instance before the first storage unit is restored. When the correlated faults involve a LSE, the situation becomes even more unlikely, requiring the other copy of the deduplicated chunk to not only reside on the same disk as the error, but the same stripe.

For those systems which did suffer unrecoverable data loss, we kept a tally of the error scenarios leading to unrecoverable loss. All witnessed data loss events involved at least one disk failure. More than 50% also contained a LSE, while less than 2% contained a UWE. The high proportion of LSEs contributing to unrecoverable data loss stems from the temporal locality described by [10]. Unrecoverable data loss usually occurred during a campaign of LSEs coupled with a drive failure in the effected RAID unit (66.9% of the time), or an failure of two drives in a RAID unit before a rebuild could be accomplished (31.8% of the time).

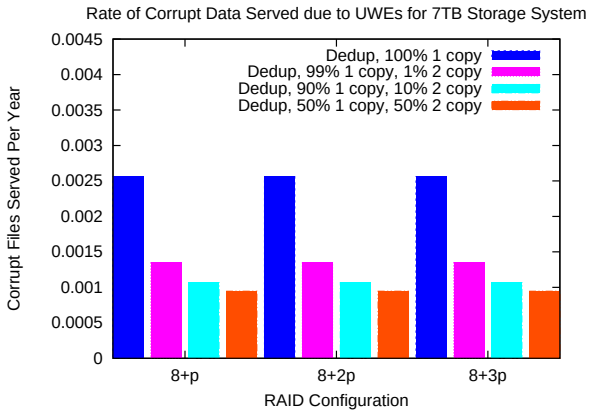
The decrease in fault tolerance due to data deduplication is easily offset by maintaining multiple copies of each deduplicated instance. Keeping as few as one additional copy results in a system more fault tolerant than the original, while still resulting in a mean increase in storage efficiency for all categories. It is important, however, to ensure additional copies are kept on separate RAID units to reduce the chance of correlated losses.

Multi-instance data deduplication maintains more than one copy for a distinct data chunk. It increases the resiliency of the system by orders of magnitude at the cost of increased space usage. The performance characteristics of such a system, i.e. write characteristics (data injection) and read characteristics (data reconstruction) are highly dependent on the architecture of the system, specifically the architecture of the meta-data manager. Unlike traditional single instance deduplication systems where hash maps or indices maintain data-signature to data-location mappings, in multi-instance deduplication systems, these bookkeeping data structures now have to accommodate more complex mappings. Further, managing (create/use/delete) these complex mappings adds overhead to both the CPU and IO.

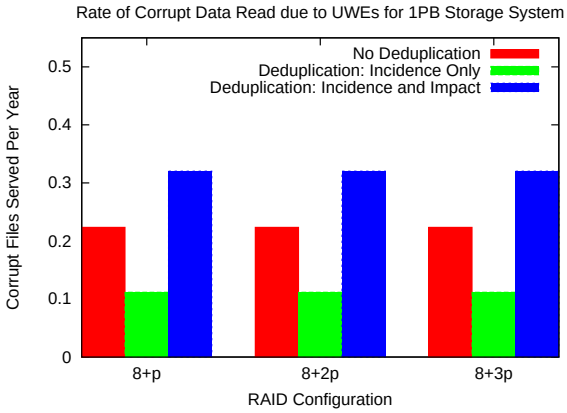
While UWEs did not significantly contribute to unrecoverable data loss, it is important to remember that they are fail silent, and largely orthogonal to RAID [40, 55]. When a UWE



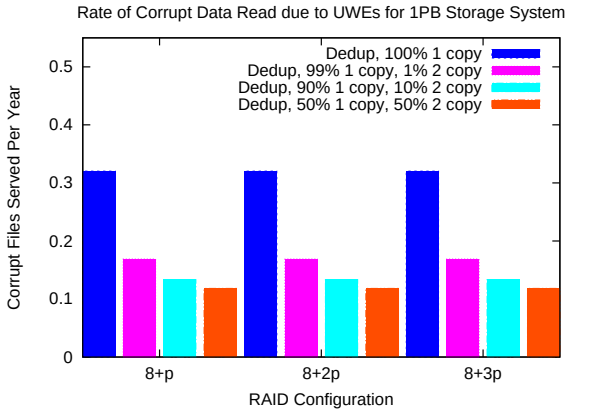
(a) Corrupted data served for systems with no deduplication, and with a single copy of each deduplicated chunk.



(b) Corrupted data served for systems with a single copy of each deduplicated chunk, and with two copies kept for the 1%, 10%, and 50% most referenced chunks.



(c) Corrupted data served for systems with no deduplication, and with a single copy of each deduplicated chunk.

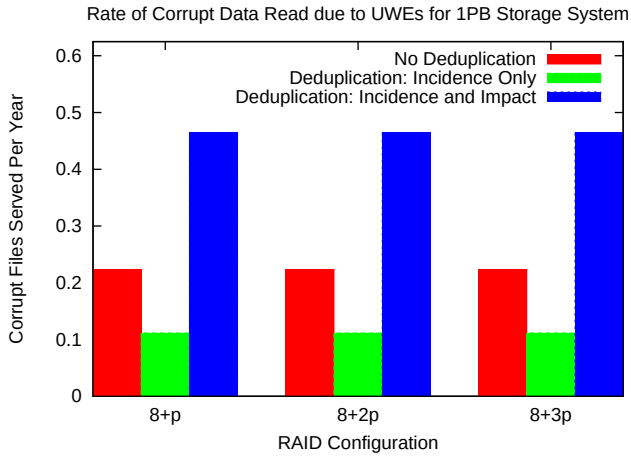


(d) Corrupted data served for systems with a single copy of each deduplicated chunk, and with two copies kept for the 1%, 10%, and 50% most referenced chunks.

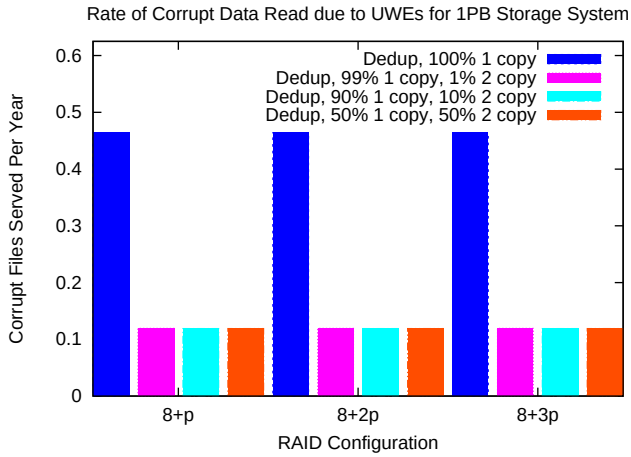
Figure 6.9: Rate of undetected corrupted reads for the SQL category, for systems with data sets of size 7TB and 1PB.

occurs on a system, it can cause corrupted data to be silently served when requested, before a disk scrub corrects the error. Figures 6.9a and 6.9d show the difference in the rate of corrupt data served for three different storage configurations for our 7TB and 1PB systems respectively for a sample data category. The first bar for each RAID configuration shows the rate of corrupt data being read for a non-deduplicated system. The second bar, to illustrate the different effects of incidence and impact, shows the different incidence only. The third bar shows the full effect of both incidence and impact. While the incidence of corrupt data is reduced, i.e. the smaller amount of data stored results in a lower number of corrupted files, the increased number of references results in a higher incidence of corrupted data being served. Not only are reads to the corrupted file effected, but any read to a referring file will result in silently serving corrupt data to the user, increasing the overall rate of corrupted reads to the system due to a UWE.

Again, we find a solution by keeping multiple copies of each deduplicated instance. Figures 6.9b and 6.9d compare the rate of corrupted data being read of deduplicated systems with a single copy of each instance, to systems which keep two copies for a fraction of all files in a category. The graph shows the results for keeping two copies for the 1%, 10% and 50% of files within a category containing the largest number of references. In the case of the SQL category, this results in large improvements for just the top 1% most referenced files. Figure 6.10 shows similar results for the VM category. While the general trends are the same, it is useful to note that the results of keeping additional copies is dependent on the category. Unlike the SQL category, increasing the portion of the category which maintains multiple copies from 1% to 10% and then again to 50% does not provide a significant reduction in the rate of corrupt data served, due to a small number of deduplicated instances accounting for a large number of the references within the category. This highlights the importance of a detailed analysis, using category information when making assumptions about the underlying deduplicated system.



(a) 1PB system, with only a single copy of each deduplicated instance.



(b) 1PB system, with multiple copies kept for some files.

Figure 6.10: Rate of undetected corrupted reads for the VM category.

6.7 Conclusions

Our evaluation of the effect of deduplication on our example system leads us to conclude that deduplication has a net negative impact on reliability, both due to its impact on unrecoverable data loss, and the impact of silent data corruptions, though the former is easily countered by using higher level RAID configurations. In both cases, system reliability can be increased by maintaining additional copies of deduplicated instances, and for the categories identified in our example system, typically by keeping multiple copies for a very small percentage of the deduplicated instances in a given category.

Our results emphasize the importance of detailed analysis of deduplicated systems to fully understand the impact of deduplication on fault tolerance. Even within our example system, individual categories features very different distributions, resulting in differing behaviors and trade-offs for multi-copy deduplication. Reliability returns decrease sharply for the VM category with increased proportions stored as multiple copies, due to the high portion with only a few references. For the VM category 90% had two or fewer references. Conversely, only 38% of deduplicated instances in the MSSQL category had two or fewer references.

While data deduplication helps to achieve goals of storage efficiency, its increasing prevalence raises legitimate reliability concerns. Given the increased regulatory pressure, and a desire to meet customer requirements for long-term data integrity, it is important to develop a further understanding of the reliability consequences of these methods.

For our example system, we show that reliability goals can still be met while maintaining some of the storage efficiency provided by deduplication by storing multiple copies of a portion of deduplicated instances. Our methodology could be applied to other systems to generate similar evaluations, and to evaluate configurations to meet design goals for both reliability and storage efficiency.

CHAPTER 7

CONCLUSIONS

As storage systems continue to grow in scale and complexity, it becomes more important to understand how they fail, and the effectiveness of various methods of preventing failure. In response to the increased amount of data that needs to be stored, and the length of time the data is expected to be stored, it is likely that new techniques for improving storage efficiency will continue to be developed Those techniques, like deduplication, are likely to have complex and potentially adverse effects on the fault tolerance of the underlying storage systems to which they are applied. Efficient methods for analyzing potential designs, and their ability to achieve reliability and storage efficiency goals, will grow more important.

To provide system designers with tools to analyze designs, we have presented models of the current faults affecting modern and next-generation storage systems, and the hardware and process layers of large-scale deduplicated file systems. We have detailed methods to analyze dependence relationships in these systems, and methods to exploit patterns that improve the efficiency of solution methods when applied to these storage systems in fault-tolerance studies. In this chapter, we briefly review the work we have presented in this dissertation and describe potential avenues for expansion of our work, before concluding with some final remarks.

7.1 Contribution Review

We first presented a detailed set of models that can affect modern storage systems, including a novel model for UDEs. The models were expanded throughout our dissertation to account for complexities present in large-scale systems, fault interaction, and the ways in which faults interact with recovery techniques. These models improve our understanding of how complex

systems fail and provide a rich environment of failures in which to conduct reliability studies.

To provide a methodology for building large-scale models we utilize the idea of component-based models using data from real systems to extrapolate models of planned systems which have not yet been built. Those methods allow us to develop a better understanding of systems being designed by studying the types of components in existing systems, the ways they interact, and dependence relationships found between components. We explore the methods by modeling a large-scale, production, clustered file system and the design of its replacement.

We next presented a method for efficient simulation of large-scale storage systems that takes advantage of the component-based approach to model individual RAID subsystems and scale an existing system to the petascale level. To solve large-scale models with many components quickly through decomposition of our model, our method exploits the regular structure of that model, the insight that dependence relationships within a RAID system matter only when faults are present, and the fact that separate RAID systems do not normally interact.

To expand the utility of our analysis of system interdependence, and our ability to exploit dependence relationships in reliability models to enable more efficient simulation, we looked at deduplicated file systems. Deduplication presents us with more complex dependence relationships that we proposed to study with MDGs, a novel structure for encoding dependence relationships in storage systems. We presented methods for identifying failure and recovery actions, along with methods for decomposing models based on their current state, and the relevance of various identified dependencies to the reward variables defined for our model.

Finally we applied our methods to a large-scale production deduplicated file system. We presented a novel method for generating models of deduplicated file systems based on empirical data, and used this model, along with a detailed system model, to answer questions about the impact of deduplication on reliability. We showed for the system analyzed that deduplication resulted in a net reduction of fault tolerance, and provided a method to achieve improved reliability by maintaining multiple references for a subset of the system data. In doing so we showed how to achieve improved reliability, along with improved storage efficiency, even over non-deduplicated systems. Our methods provide more fault coverage than

RAID systems do, by allowing us to identify UDEs, which are normally orthogonal to RAID.

7.2 Future Work

We have identified three primary avenues of future work. In Section 7.2.1, we discuss plans to expand existing fault models to allow for solid state storage media, a type of storage medium for which fault tolerance is poorly understood. In Section 7.2.2, we explain ways to extend our work to account for the use of deduplication in primary storage applications, and the unique design challenges that need to be explored to extend our proposed fault-tolerance methods to this domain. In Section 7.2.3 we discuss methods for using the information provided by a deduplication server with or without actually deduplicating data to guard against UDEs and other difficult faults in primary storage systems.

7.2.1 Solid-State Disks

Models for SSD reliability are still in their infancy, but some do exist. Failures in SSDs are usually modeled as a bit error rate, and more importantly as an uncorrected bit error rate (UBER). Common sources of errors include program disturb (due to tunneling or hot-electron injection), quantum noise effects, erratic tunneling, SILC-related data retention, read disturb, and detrapping-induced retention. These can be broken down into three major categories: write errors, retention errors, and read-disturb. [75]

A basic type of write error is the *program disturb*. Program disturb occurs when a cell becomes unintentionally programmed due to the programming of another cell. *Overprogramming* due to noise, tunneling, or other reasons is another possibility, causing a cell that is being programmed to receive an incorrect number of pulses. A cell being programmed can also suffer from a program disturb in such a way that it results in overprogramming. [76, 77, 78, 79, 80, 81, 82, 83, 84]

Retention errors are caused by stresses outside of the program/erase cycle, and affect the ability of a cell to retain data over time. Bit errors due to retention problems are usually related to charge loss, moving from one voltage threshold to another; the primary causes of

which are stress-induced leakage current (SILC) and detrapping of tunnel-oxide charge that was trapped during cycling. [85, 86, 87, 88, 89, 90, 91, 92, 93, 94]

Read-disturb errors represent errors which occur due to interference during a read operation. When a cell is read, all deselected wordlines in the block have a voltage applied. This can disturb bits either through SILC, or through the filling of traps in the tunnel oxide. [85, 86, 87, 88, 89, 90, 91, 92, 93, 94]

Reasonable models for the UBER that results from these effects are given in the literature, along with information of how they interact. [75] [95]

Failure rates, however, have been shown not to be stationary, and more complex models are needed to account for the age of NAND memory and how this variable affects RBER. Distributions of NAND memory age within a storage system are also necessary.

7.2.2 Primary Storage Deduplication

We have shown the application of our methods to archival storage systems, but deduplication has been increasingly used for primary storage as well. Primary storage adds the challenge of requiring a much higher standard of performance than archival storage does. In order to apply our multi-copy deduplication methods to deduplicated primary storage systems, we would need to use our fault tolerance results to inform a performance model of a primary storage system to assess the impact of multi-copy deduplication on performance measures of interest for various systems.

We plan to investigate those systems, and to develop an understanding of the differences between the underlying models of deduplication-based dependence relationships in primary storage systems, versus archival systems. Since primary storage tends to be more diverse than archival storage, it is likely that the deduplication ratio will be lower, potentially requiring other methods to retain storage efficiency when multi-copy deduplication is used.

7.2.3 UDE-Tolerant Storage Systems

We have discussed the use of multi-copy deduplication in storage systems to mitigate the negative impact of deduplication on fault tolerance. However, another potential use of our methods includes non-deduplicated systems. It is possible to develop a method for counteracting UDEs that uses the information provided by a deduplication server about redundancy in a storage system, without completely eliminating that redundancy. UDEs represent a real challenge as detecting them has traditionally been too expensive, given their orthogonality to RAID. However, given proper file placement to ensure that redundant information can be read in parallel, deduplication meta-data could be used to check on read operations for corrupt, stale, or otherwise incorrect data due to a UDE.

In order to assess the suitability of those methods, we plan to analyze their performance consequences in the presence of real workloads for both primary and secondary storage, to assess the reduction in throughput required to perform these checks with enough frequency to meet reliability goals for data despite the presence of UDEs.

7.3 Concluding Remarks

As storage technology companies continue to expand the horizon of storage technology, it is critical that we also continue to expand our understanding of the systems in question. As systems scale-up, faults that once had no noticeable effect become common, and can threaten the integrity of system data. New technologies that are intended to address certain customer requirements often have complex effects on other requirements that are not well understood. Our understanding of the fault environments present in next-generation storage systems and their complex inter-dependencies needs to keep pace with technology.

Work in this area has often relied on incomplete models of the extant fault environments and incomplete models of the underlying file relationships in storage systems have led to incorrect or incomplete conclusions about the effect of the application of deduplication to storage systems. Our goal at the outset of the research that led to this dissertation was to develop realistic models based on data from real systems to improve our understanding of

the fault tolerance characteristics of modern storage systems. That entailed derivation of new models of emerging faults, models of dependence relationships in the stored data, and models of the important characteristics of the hardware portions of storage systems.

The models we created proved to be large, and, due to the stiffness of the underlying systems, difficult to analyze. Our early efforts to decompose simple, non-deduplicated systems, while successful, lacked the expressiveness to handle complex dependence relationships. We thus set out to develop methods to analyze those more complex relationships, and exploited the characteristics we found in reliability models of deduplicated storage systems to improve the efficiency of our solution methods, which we then applied to a real production system. That enabled us to improve our understanding of the fault-tolerance properties of modern systems, and to propose new ways to improve fault-tolerance that has been degraded by deduplication.

There is still a wide range of unsolved problems in storage modeling. Further models need to be developed to handle nonmagnetic media, such as SSDs, and newer technologies on the horizon, such as phase-change memory. As these new types of storage become more affordable, they will see increased use in production systems. Already, solid-state devices fulfill the local storage requirements of mobile devices and many laptops. Next-generation storage systems are already being proposed that integrate magnetic and solid-state storage. We believe our approach provides practical models and methods to tackle current storage system design problems, and provides a foundation upon that to develop new research which has the potential to provide understanding of even more complex systems, allowing system architects to design, understand, and deploy reliable and efficient storage systems despite the many challenges inherent in such systems.

APPENDIX A

EXAMPLE IDENTIFICATION OF FAULT-DEPENDENT RARE EVENTS

In this appendix, we present the results of the application of our algorithm from Chapter 5 to a model of the state of a block from the system described in Chapter 6.

Figure A.1 shows a partial model of a single block from the system from Chapter 6. Four state variables are shown in this representation:

- `scrub_process` - A state variable that holds the scrub token, indicating it is available to initiate a scrub of the disk associated with this block.
- `block_state` - The state of the current block; the value of this state variable indicates whether the block has suffered from a fault, and the type of the fault.
- `parity_state` - The state of the parity block on another disk in the RAID group, associated with this block.
- `LSE_count` - A counter that holds a number of LSEs that can affect blocks in proximity to the current block, should it suffer an initial LSE.

Ten events are also shown in Figure A.1:

- `advance_scrub` - Two events with this label are shown. They serve to move the scrub token between disks in a loop, as the scrub process examines each disk in the larger storage system in turn.
- `scrub` - This event fires only when a scrub-detectable state (LSE or UDE) is present in `block_state`, and the scrub token is present in `scrub_process`. When firing, it corrects an LSE or UDE in `block_state`.

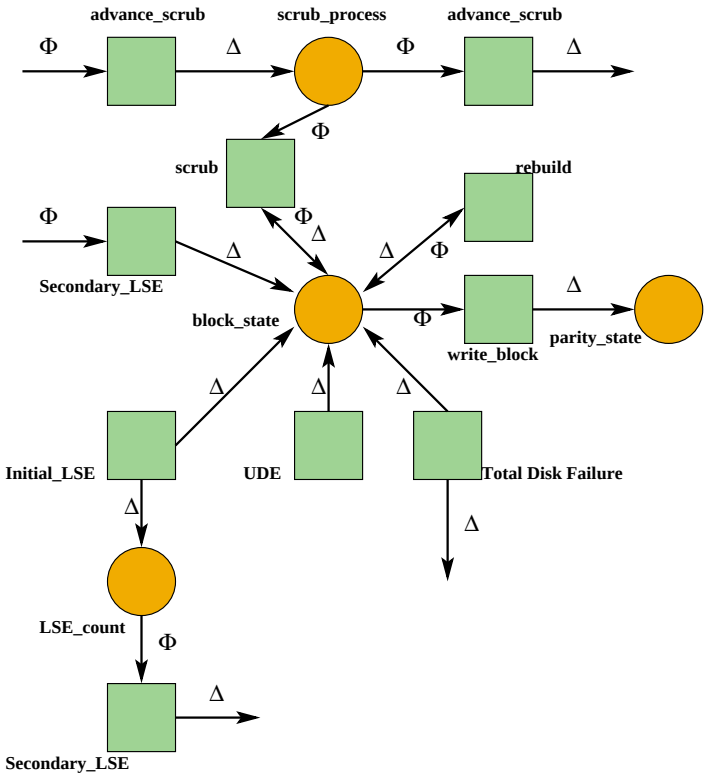


Figure A.1: Initial block model

- `rebuild` - This event only fires when `block_state` indicates that the block is part of a disk that has suffered a whole disk failure. It represents the RAID rebuild action that corrects the state, and it affects all blocks on the disk.

- `write_block` - This event only fires when `block_state` indicates that a UDE has occurred, and changes the state of `block_state` and `parity_state` to indicate parity pollution has occurred.

- `Initial_LSE` - This event represents the occurrence of the first of a series of LSEs on a given disk, originating in the sector containing the modeled block.

- `Secondary_LSE` - Two events share this label. The first represents secondary LSEs that occur when `LSE_count` > 0; the second representing a similar action from another block spatially close to the modeled block.

- `UDE` - This event represents the occurrence of a UDE in our modeled block.

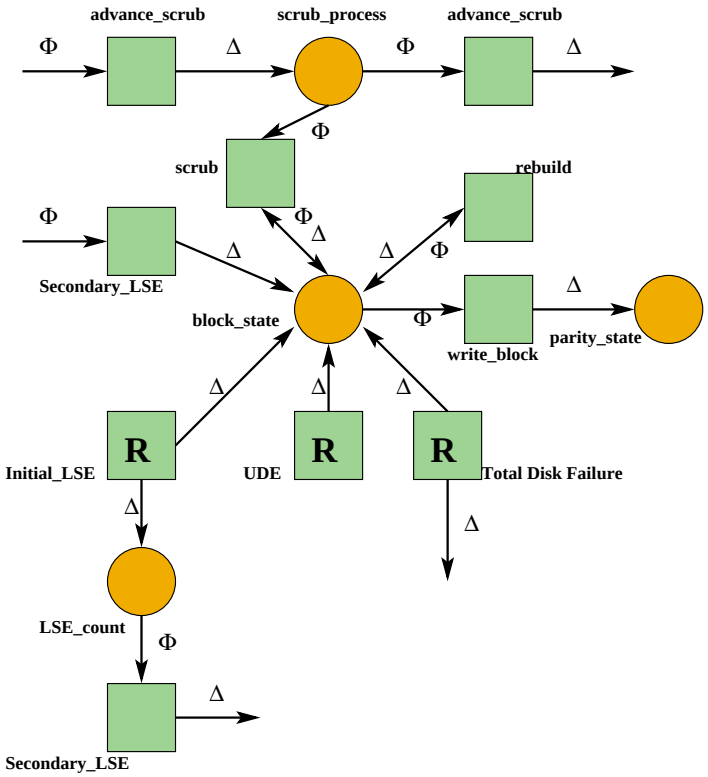


Figure A.2: Block model with initial rare events indicated.

- `Total_Disk_Failure` - This event represents the failure of the entire disk that contains the modeled block.

We begin the process by marking those events in our model that are locally rare due to their rates, in this case `Initial_LSE`, `UDE`, and `Total_Disk_Failure`, as shown in Figure A.2.

We then process the model using Algorithm 1 and assuming a system with no initial faults. The first step of the algorithm is to remove all Δ-dependencies from rare events in E_R, shown in Figure A.3.

Next, we mark as constant any state variable whose value can no longer change, except for the firing of an event in E_R. `LSE_count` fits this category, having no remaining Δ-dependencies, and so is marked constant with a value of 0. Likewise, the equivalent state variable is also marked constant for the incoming secondary LSEs to our block model. The result is shown in Figure A.4.

We then remove all Φ-dependencies that cannot be satisfied by the constant marking of

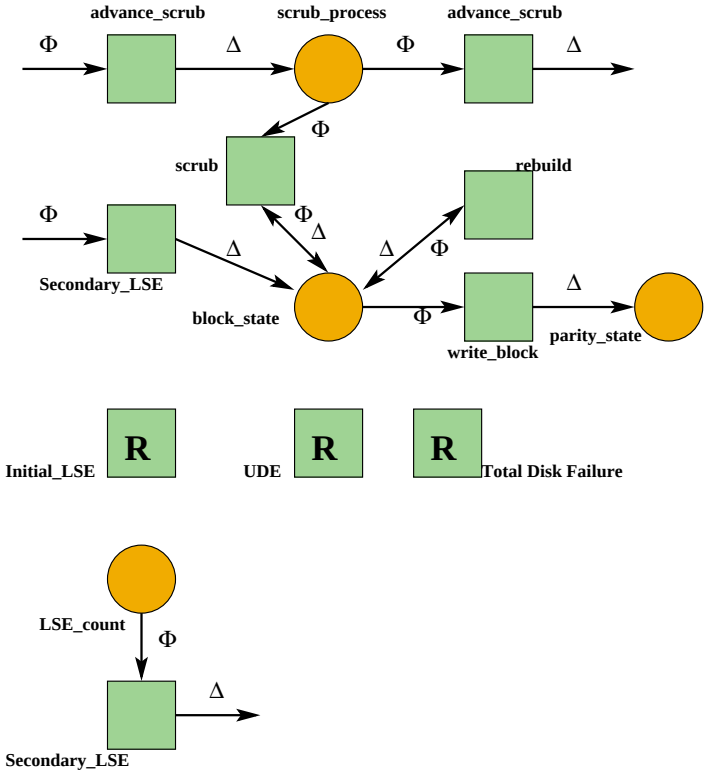


Figure A.3: Block model, with Δ-dependencies eliminated.

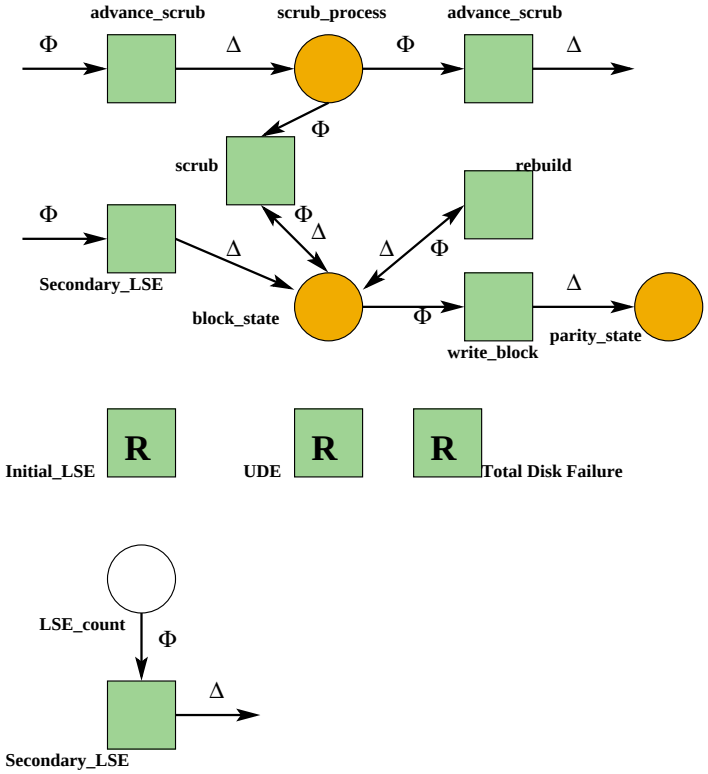


Figure A.4: Block model with state variables marked constant.

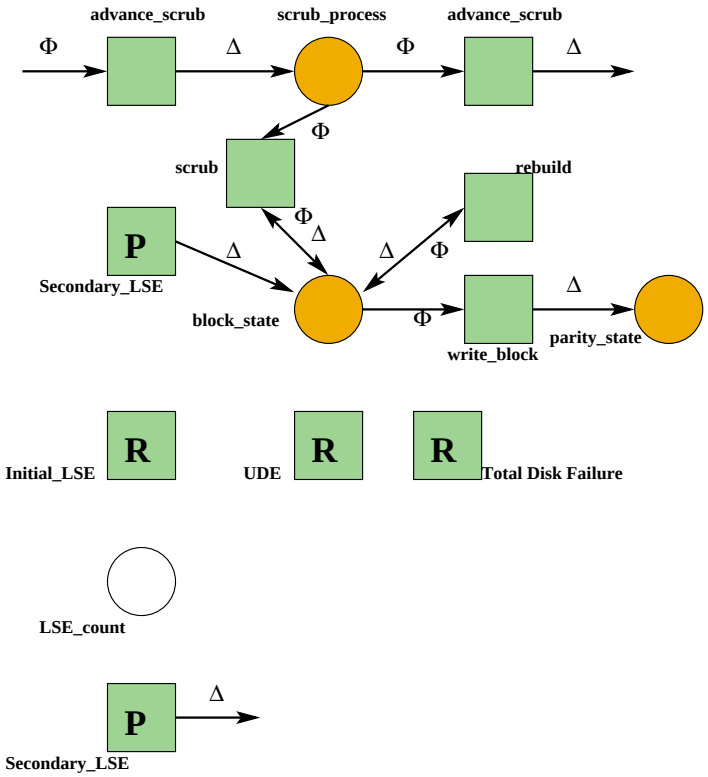


Figure A.5: Block model with Φ-dependencies removed, and new events labeled in P.

state variables, and take those events whose enabling conditions can no longer be satisfied and place them in the set P, as shown in Figure A.5.

We repeat the previous step of removing Δ-dependencies from events in P, which we now add to the set E_R, shown in Figure A.6.

Again we mark as constant those state variables whose values can no longer change, in this case `block_state`, as shown in Figure A.7.

We remove all Φ-dependencies that cannot be satisfied by the constant marking of state variables, and take those events whose enabling conditions can no longer be satisfied and place them in the set P, as shown in Figure A.8.

We repeat the previous step of removing Δ-dependencies from events in P, which we now add to the set E_R, as shown in Figure A.9.

At this point no further events can be removed, and the algorithm halts, having identified our mitigation actions and added them to E_R.

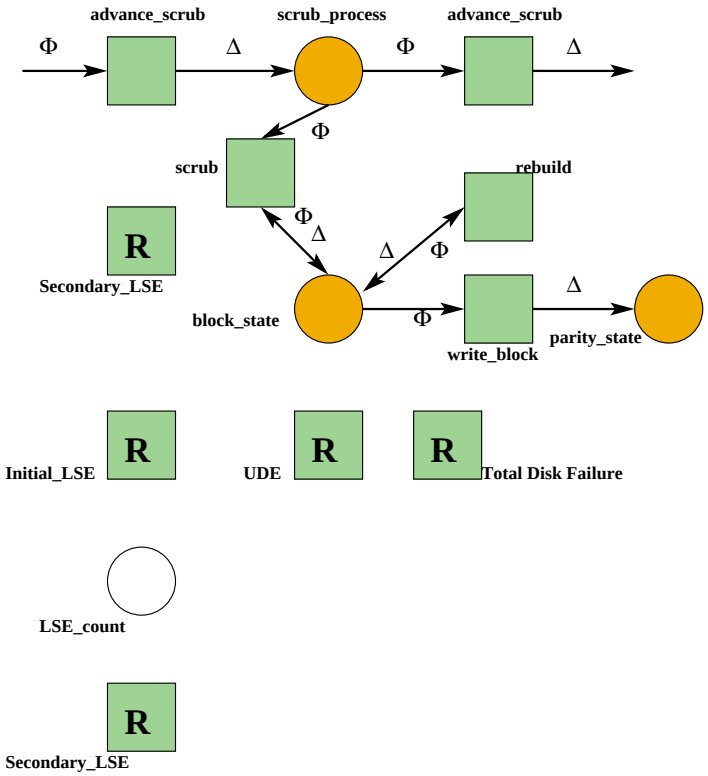


Figure A.6: Block model, with Δ-dependencies eliminated.

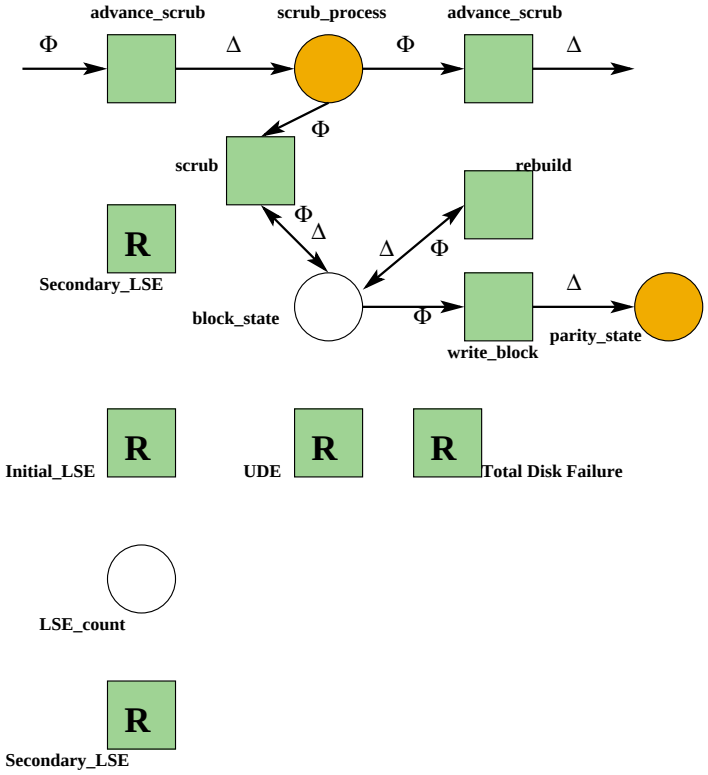


Figure A.7: Block model with state variables marked constant.

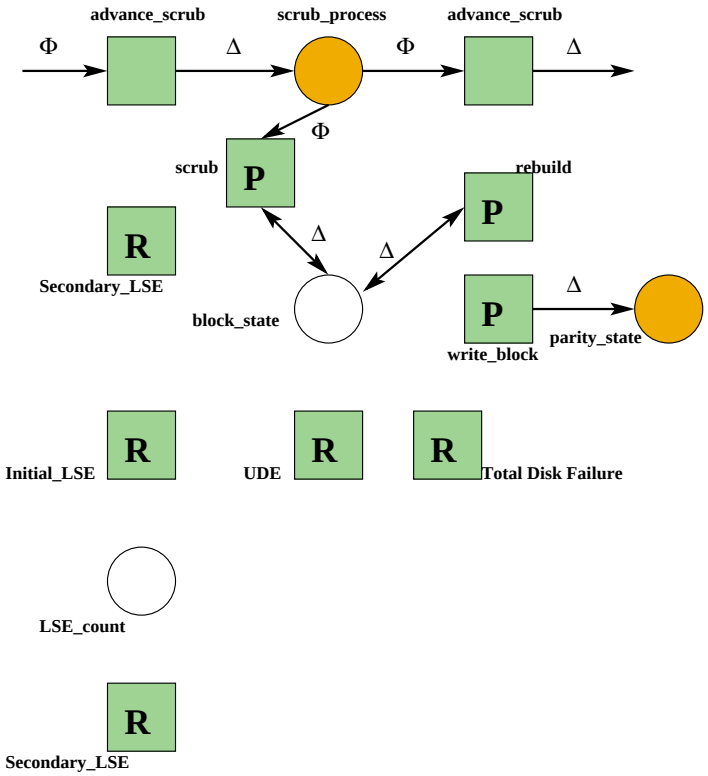


Figure A.8: Block model with Φ-dependencies removed, and new events labeled in P.

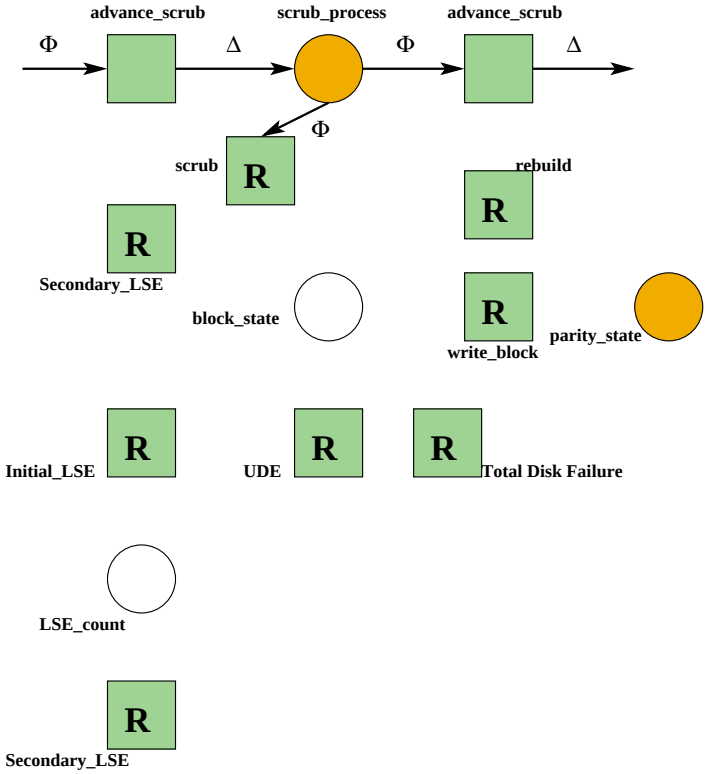


Figure A.9: Final block model with all mitigation actions identified.

REFERENCES

[1] J. F. Gantz, C. Chute, A. Manfrediz, S. Minton, D. Reinsel, W. Schlichting, and A. Toncheva, "The diverse and exploding digital universe: An updated forecast of worldwide information growth through 2011," White Paper, IDC, March 2008.

[2] P. Lyman, H. R. Varian, K. Searingen, P. Charles, N. Good, L. L. Jordan, and J. Pal, "How much information?" 2003. [Online]. Available: http://www.sims.berkeley.edu/research/projects/how-much-info/

[3] J. F. Gantz and D. Reinsel, "Extracting value from chaos," White Paper, IDC, June 2011.

[4] "Compliance: The effect on information management and the storage industry." 2003. [Online]. Available: http://www.enterprisestoragegroup.com/

[5] L. L. You, K. T. Pollack, and D. D. E. Long, "Deep store: An archival storage system architecture," in *ICDE*. IEEE, 2005, pp. 804–815.

[6] J. Gray, P. Shenoy, J. Gray, P. Shenoy, and P. Shenoy, "Rules of thumb in data engineering," in *Proceedings of the 16th International Conference on Data Engineering*. Washington, DC, USA: IEEE Computer Society, 2000. [Online]. Available: http://dl.acm.org/citation.cfm?id=846219.847318 pp. 3–12.

[7] J. G. Elerath, "Specifying reliability in the disk drive industry: No more MTBFs," in *In Proc. of the Annual Reliability and Maintainability Symposium*, 2000.

[8] B. Schroeder and G. A. Gibson, "Understanding disk failure rates: What does an MTTF of 1,000,000 hours mean to you?" *Trans. Storage 3*, no. 3, p. 8, 2007.

[9] L. N. Bairavasundaram, G. R. Goodson, S. Pasupathy, and J. Schindler, "An analysis of latent sector errors in disk drives," *SIGMETRICS 35*, no. 1, pp. 289–300, 2007.

[10] B. Schroeder, S. Damouras, and P. Gill, "Understanding latent sector errors and how to protect against them," in *FAST*, 2010, pp. 71–84.

[11] D. A. Reed, C. da Lu, and C. L. Mendes, "Reliability challenges in large systems," *Future Generation Computer Systems 22*, 2006.

[12] D. A. Patterson, G. A. Gibson, and R. H. Katz, "A case for Redundant Arrays of Inexpensive Disks (RAID)," EECS Department, UC Berkeley, Tech. Rep. UCB/CSD-87-391, 1987. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/1987/5853.html

[13] L. N. Bairavasundaram, G. R. Goodson, B. Schroeder, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dussea, "An analysis of data corruption in the storage stack," in *FAST*. USENIX, 2008, pp. 1–16.

[14] B. Zhu, K. Li, and H. Patterson, "Avoiding the disk bottleneck in the data domain deduplication file system," in *USENIX FAST*, 2008, pp. 1–14.

[15] C. Ungureanu, B. Atkin, A. Aranya, S. Gokhale, S. Rago, G. Calkowski, C. Dubnicki, and A. Bohra, "HydraFS: A high-throughput file system for the HYDRAstor content-addressable storage system," in *FAST*, 2010, pp. 225–238.

[16] D. Bhagwat, K. Pollack, D. D. E. Long, T. Schwarz, E. L. Miller, and J.-F. Pris, "Providing high reliability in a minimum redundancy archival storage system," in *IEEE MASCOTS*, 2006, pp. 413–421.

[17] M. Ajtai, R. Burns, R. Fagin, D. D. E. Long, and L. Stockmeyer, "Compactly encoding unstructured inputs with differential compression," *JACM 49*, pp. 318–367, 2002.

[18] F. Douglis and A. Iyengar, "Application-specific delta-encoding via resemblance detection," 2003.

[19] F. Douglis and A. Iyengar, "Application-specific delta-encoding via resemblence detection," in *USENIX ATEC*, 2003.

[20] L. You and C. Karamanolis, "Evaluation of efficient archival storage techniques," in *IEEE/NASA Goddard MSST*, 2004.

[21] J. MacDonald, "File system support for delta compression," M.S. thesis, UC, Berkeley, 2000.

[22] L. Freeman, "How safe is deduplication," NetApp, Tech. Rep., 2008. [Online]. Available: http://media.netapp.com/documents/tot0608.pdf

[23] "NCSA Intel 64 linux cluster Abe technical summary http://www.ncsa.uiuc.edu/UserInfo/Resources/Hardware/Intel64Cluster/TechSummary/," July 9 2007.

[24] E. Strohmaier, J. J. Dongarra, H. W. Meuer, and H. D. Simon, "The marketplace of high performance computing," *Parallel Computing*, vol. 25, no. 13–14, pp. 1517–1544, 1999. [Online]. Available: citeseer.ist.psu.edu/strohmaier99marketplace.html

[25] F. Schmuck and R. Haskin, "GPFS: A shared-disk file system for large computing clusters," in *In Proc. File And Storage Technologies, FAST*, 2002, pp. 231–244.

[26] W. Yu, S. Liang, and D. K. Panda, "High performance support of parallel virtual file system (PVFS2) over Quadrics," in *International Conference on Supercomputing, ICS*, 2005, pp. 323–331.

[27] R. L. Braby, J. E. Garlick, and R. J. Goldstone, "Achieving order through CHAOS: The LLNL HPC Linux Cluster Experience," in *The 4th International Conference on Linux Clusters: The HPC Revolution 2003*, San Jose, CA, USA, 2003.

[28] Y. Liang, Y. Zhang, A. Sivasubramaniam, M. Jette, and R. Sahoo, "BlueGene/L Failure Analysis and Prediction Models," in *Dependable Systems and Networks, DSN*, 2006, pp. 425–434.

[29] L. Wang, K. Pattabiraman, Z. Kalbarczyk, R. K. Iyer, L. Votta, C. Vick, and A. Wood, "Modeling coordinated checkpointing for large-scale supercomputers," in *Dependable Systems and Networks, DSN*, 2005, pp. 812–821.

[30] B. Schroeder and G. A. Gibson, "A large-scale study of failures in high-performance computing systems," in *Dependable Systems and Networks, DSN*, 2006, pp. 249–258.

[31] D. Tang and R. K. Iyer, "Dependability measurement and modeling of a multicomputer system," *IEEE Transactions on Computers*, vol. 42, no. 1, pp. 62–75, 1993.

[32] B. Schroeder and G. A. Gibson, "Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you?" in *File And Storage Technologies, FAST*, 2007, pp. 1–16.

[33] W. Jiang, C. Hu, Y. Zhou, and A. Kanevsky, "Are disks the dominant contributor for storage failures? A comprehensive study of storage subsystem failure characteristics," in *File And Storage Technologies, FAST*, 2008, pp. 111–125.

[34] E. Rosti, G. Serazzi, E. Smirni, and M. S. Squillante, "Models of parallel applications with large computation and I/O requirements," *IEEE Transactions in Software Engineering*, vol. 28, no. 3, pp. 286–307, 2002.

[35] "Notifications to Teragrid users about CFS unavailability," Tech. Rep., November 2007. [Online]. Available: http://news.teragrid.org/user.php?cat=ncsa

[36] L. McBryde, G. Manning, D. Illar, R. Williams, and M. Piszczek, "Data Management Architecture," US Patent number 7127668, Oct 24, 2006.

[37] D. W. Hosmer and S. Lemeshow, *Applied Survival Analysis: Regression Modeling of Time to Event Data*. New York, NY, USA: John Wiley & Sons, Inc., 1999.

[38] G. Clark, T. Courtney, D. Daly, D. Deavours, S. Derisavi, J. M. Doyle, W. H. Sanders, and P. Webster., "The Möbius modeling tool." in *Proceedings of the 9th International Workshop on Petri Nets and Performance Models*, 2001, pp. 241–250.

[39] B. Schroeder and G. A. Gibson, "Understanding failures in petascale computers," *J of Physics 78*, 2007.

[40] J. L. Hafner, V. Deenadhayalan, W. Belluomini, and K. Rao, "Undetected disk errors in RAID arrays," *IBM J Research and Development 52*, no. 4, pp. 413–425, 2008.

[41] B. Schroeder and G. A. Gibson, "Disk failures in the real world: what does an MTTF of 1,000,000 hours mean to you?" in *FAST*, 2007, p. 1.

[42] J. G. Elerath, "AFR: problems of definition, calculation and measurement in a commercial environment," in *In Proc. of the Annual Reliability and Maintainability Symposium*, 2000.

[43] J. G. Elerath and S. Shah, "Server class disk drives: How reliable are they?" in *In Proc. of the Annual Reliability and Maintainability Symposium*, 2004, pp. 151–156.

[44] J. Gray and C. van Ingen, "Empirical measurements of disk failure rates and error rates." in *Technical Report MSR-TR-2005-166*, 2005.

[45] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Iron file systems," *ACM SIGOPS 39*, no. 5, pp. 206–220, 2005.

[46] J. L. Hafner, V. Deenadhayalan, K. Rao, and J. A. Tomlin, "Matrix methods for lost data reconstruction in erasure codes," in *USENIX FAST*, 2005, pp. 15–30.

[47] A. Krioukov, L. N. Bairavasundaram, G. R. Goodson, K. Srinivasan, R. Thelen, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dussea, "Parity lost and parity regained," in *FAST*. USENIX, 2008, pp. 1–15.

[48] P. Kelemen, "Silent corruptions," Tech. Rep., June 2007. [Online]. Available: http://fuji.web.cern.ch/fuji/talk/2007/kelemen-2007-C5-SilentCorruptions.pdf

[49] E. M. Evans, "Working Draft Project, American National Standard T10/1799-D, "Information Technology – SCSI Block Commands-3 (SBC-3)", May, 13, 2008," http://www.t10.org/ftp/t10/drafts/sbc3/sbc3r15.pdf.

[50] B. E. Clark, F. D. Lawlor, W. E. Schmidt-Stumpf, T. J. Stewart, and G. D. T. Jr., "Parity spreading to enhance storage access," US Patent No. 4761785, 1988.

[51] J. Keilson, *Markov Chain Models–Rarity and Exponentiality*. Springer, 1979.

[52] J. Banks, J. Carson, B. L. Nelson, and D. Nicol, *Discrete-Event System Simulation*. Prentice-Hall, 2004.

[53] I. Iliadis, R. Haas, X.-Y. Hu, and E. Eleftheriou, "Disk scrubbing versus intra-disk redundancy for high-reliability RAID storage systems," in *ACM SIGMETRICS*. Annapolis, MD, USA: ACM, 2008, pp. 241–252.

[54] M. H. Darden, "Data integrity: The dell emc distinction," Tech. Rep., May 2002. [Online]. Available: http://www.dell.com/content/topics/global.aspx/power/en/ps2q02darden?c=us&cs=555&l=en&s=biz

[55] E. W. D. Rozier, W. Belluomini, V. Deenadhayalan, J. Hafner, K. K. Rao, and P. Zhou, "Evaluating the impact of undetected disk errors in RAID systems," in *DSN*, 2009, pp. 83–92.

[56] P. Shahabuddin, "Importance sampling for the simulation of highly reliable Markovian systems," *Manage. Sci. 40*, pp. 333–352, 1994.

[57] P. W. Glynn and D. L. Iglehart, "Importance sampling for stochastic simulations," *Manage. Sci. 35*, pp. 1367–1392, 1989.

[58] H. Kahn and T. E. Harris, "Estimation of particle transmission by random sampling," in *NBS AMS*, 1951.

[59] M. J. J. Garvels, "The splitting method in rare event simulation," Ph.D. dissertation, Univ. of Twente, Enschede.

[60] P. J. Courtois, *Decomposability: Queueing and Computer System Applications*. New York: Academic Press, 1977.

[61] K. S. Trivedi and R. M. Geist, "Decomposition in reliability analysis of fault-tolerant systems," *Reliability, IEEE Trans. on*, vol. R-32, no. 5, pp. 463 –468, 1983.

[62] W. D. Oball II, *Measure-Adaptive State-Space Construction Methods*. U Arizona, 1998.

[63] W. Sanders and J. Meyer, "A unified approach for specifying measures of performance, dependability, and performability," in *DCCA 4*. Springer, 1991, pp. 215–237.

[64] R. A. Howard, *Dynamic Probabilistic Systems. Vol II: Semi-Markov and Decision Processes*. New York: Wiley, 1971.

[65] J. F. Meyer, "On evaluating the performability of degradable computing systems," *IEEE TC 29*, pp. 720–731, 1980.

[66] W. H. Sanders and J. F. Meyer, "A unified approach to specifying measures of performance, dependability, and performability," *Dependable Computing for Critical Applications*, vol. 4, pp. 215–237, 1991.

[67] G. Ciardo and K. S. Trivedi, "A decomposition approach for stochastic reward net models," *Performance Eval. 18*, no. 1, pp. 37 – 59, 1993.

[68] T. R. Kiehl, R. M. Mattheyses, and M. K. Simmons, "Hybrid simulation of cellular behavior," *Bioinformatics 20*, pp. 316–322, 2004.

[69] H. Salis and Y. Kaznessis, "Accurate hybrid stochastic simulation of a system of coupled chemical or biochemical reactions," *J Chemical Physics 122*, no. 5, 2005.

[70] J. Bucklew and R. Radeke, "On the Monte Carlo simulation of digital communication systems in Gaussian noise," *IEEE Trans. Comm. 51*, no. 2, pp. 267 – 274, 2003.

[71] E. W. D. Rozier and W. H. Sanders, "Dependency-based decomposition of systems involving rare events," Coordinated Science Laboratory, Univ. of Illinois, Tech. Rep. UILU-ENG-11-2203-CRHC-11-03, 2011.

[72] M. O. Rabin, "Fingerprinting by random polynomials," Tech. Rep., 1981.

[73] A. Z. Broder, "Identifying and filtering near-duplicate documents," in *CPM*. Springer, 2000, pp. 1–10.

[74] M. Baker, M. Shah, D. S. H. Rosenthal, M. Roussopoulos, P. Maniatis, T. Giuli, and P. Bungale, "A fresh look at the reliability of long-term digital storage," in *EuroSys, ACM SIGOPS*. ACM, 2006, pp. 221–234.

[75] N. Mielke, T. Marquart, N. Wu, J. Kessenich, H. Belgal, E. Schares, F. Trivedi, E. Goodness, and L. R. Nevill, "Bit error rate in nand flash memories," in *Reliability Physics Symposium, 2008. IRPS 2008. IEEE International*, July 2008, pp. 9–19.

[76] S. H. Hur, J. D. Lee, M. C. Park, J. D. Choi, K. C. Park, K. T. Kim, and K. N. Kim, "Effective program inhibition beyond 90nm nand flash memories," in *Proc. NVSM*, 2004, pp. 44–45.

[77] J. Lee, C. Lee, M. Lee, H. Kim, K. Park, and W. Lee, "A new program disturbance phenomenon in NAND flash memory by source/drain hot-electrons generated by GIDL current," in *IEEE NVSM*, 2006, pp. 31–33.

[78] S. Joo, H. Yang, K. Noh, H. Lee, W. Woo, J. Lee, M. Lee, W. Choi, K. Hwangh, Y. Kim, S. Sim, S. Kim, H. Chang, and G. Bae, "Abnormal disturbance mechanism of sub-100 nm nand flash memory," *Japanese J. Applied Physics*, vol. 45, pp. 6210–6215, 2006.

[79] H. Kurata, K. Otsuga, A. Kotabe, S. Kajiyama, T. Osabe, Y. Sasago, S. Narumi, K. Tokami, S. Kamohara, and O. Tsuchiya, "The impact of random telegraph signals on the scaling of multilevel flash memories," in *IEEE Symp. VLSI Circuits*, 2006, pp. 112–113.

[80] C. Compagnoni, A. Spinelli, R. Gusmeroli, A. Lacaita, S. Beltrami, A. Ghetti, and A. Visconti, "First evidence for injection statistics accuracy limitations in nand flash constant-current fowler-nordheim programming," in *IEDM Tech Dig.*, 2007, pp. 165–168.

[81] T. Ong, A. Fazio, N. Mielke, S. Pan, N. Righos, G. Atwood, , and S. Lai, "Erratic erase in etox flash memory array," in *VLSI Tech. Symp.*, 1993, pp. 83–84.

[82] A. Chimenton and P. Olivo, "Erratic erase in flash memories-part i: Basic experimental and statistical characterization," *IEEE Trans. Elect. Dev.*, vol. 50, pp. 1009–1014, April 2003.

[83] A. Brand, K. Wu, S. Pan, and D. Chin, "Novel read disturb failure mechanism induced by flash cycling."

[84] H. Belgal, N. Righos, I. Kalastirsky, J. Peterson, R. Shiner, and N. Mielke, "A new reliability model for post-cycling charge retention of flash memories," in *Proc. IRPS*, 2002, pp. 7–20.

[85] M. Kato, N. Miyamoto, H. Kume, A. Satoh, T. Adachi, M. Ushiyama, and K. Kimura, "Read-disturb degradation mechanism due to electron trapping in the tunnel oxide for low-voltage flash memories," in *IEDM Tech. Dig.*, 1994.

[86] R. Yamada, Y. Mori, Y. Okuyama, J. Yugami, T. Nishimoto, and H. Kume, "Analysis of detrap current due to oxide traps to improve flash memory retention," in *Proc. IRPS*, 2000, pp. 200–204.

[87] R. Yamada, T. Sekiguchi, Y. Okuyama, J. Yugami, and H. Kume, "A novel analysis method of threshold voltage shift due to detrap in a multi-level flash memory," in *Tech. Dig. VLSI Tech. Symp.*, 2001, pp. 115–116.

[88] J. Lee, J. Choi, D. Park, , and K. Kim, "Degradation of tunnel oxide by fn current stress and its effects on data retention characteristics of 90-nm nand flash memory," in *IRPS*, 2003, p. 497.

[89] N. Mielke, H. Belgal, I. Kalastirsky, P. Kalavade, A. Kurtz, Q. Meng, N. Righos, , and J. Wu, "Flash eeprom threshold instabilities due to charge trapping during program/erase cycling," *IEEE trans. Dev. and Mat. Reliability*, vol. 2, pp. 335–344, 2004.

[90] N. Mielke, H. Belgal, A. Fazio, Q. Meng, and N. Righos, "Recovery effects in the distributed cycling of flash memories," in *Proc. IRPS*, 2006, pp. 29–35.

[91] K. Takeuchi, s. Satoh, T. Tanaka, K. Imamiya, and K. Sakui, "A negative vth cell architecture for highly scalable, excellently noise-immune, and highly reliable nand flash memories," *IEEE J. Sol. St. Circuits*, vol. 34, pp. 675–684, May 1999.

[92] J. Gray and C. van Ingen, "Empirical measurements of disk failure rates and error rates," *Microsoft Research Technical Report MSR-TR-2005-166*, December 2005.

[93] I. S. R2-98, "Specification of hard disk drive reliability," *Stress-Test-Driven Qualification of Integrated Circuits, JEDEC Solid State Technology Association*, January 2007.

[94] J. Lee, S. Hur, and J. Choi, "Effects of floating-gate interference on nand flash memory cell operation," *IEEE Electron Device Letters*, vol. 23, pp. 265–266, May 2002.

[95] A. Kadav, M. Balakrishnan, V. Prabhakaran, and D. Malkhi, "Differential raid: Rethinking raid for ssd reliability," *SIGOPS Oper. Syst. Rev.*, vol. 44, pp. 55–59, March 2010.