Carnegie Mellon University
Research Showcase @ CMU

Parallel Data Laboratory | Research Centers and Institutes

5-2008

Using Utility Functions to Control a Distributed Storage System (CMU-PDL-08-102)

John D. Strunk
Carnegie Mellon University

Follow this and additional works at: http://repository.cmu.edu/pdl

Recommended Citation
.

This Technical Report is brought to you for free and open access by the Research Centers and Institutes at Research Showcase @ CMU. It has been accepted for inclusion in Parallel Data Laboratory by an authorized administrator of Research Showcase @ CMU. For more information, please contact research-showcase@andrew.cmu.edu.

Using Utility Functions to Control a Distributed Storage System

John D. Strunk

May 2008
CMU–PDL–08–102

Dept. of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Thesis committee

Prof. Christos Faloutsos (Carnegie Mellon University)
Prof. Gregory R. Ganger, Chair (Carnegie Mellon University)
Dr. Jeffrey O. Kephart (IBM Research)
Dr. John Wilkes (Hewlett-Packard Labs)

© 2008 John D. Strunk

Abstract

Provisioning, and later optimizing, a storage system involves an extensive set of trade-offs between system metrics, including purchase cost, performance, reliability, availability, and power. Previous work has tried to simplify provisioning and tuning tasks by allowing a system administrator to specify goals for various storage metrics. While this helps by raising the level of specification from low-level mechanisms to high-level storage system metrics, it does not permit trade-offs between those metrics.

This dissertation goes beyond goal-based requirements by allowing the system administrator to use a utility function to specify his objectives. Using utility, both the costs and benefits of configuration and tuning decisions can be examined within a single framework. This permits a provisioning system to make automated trade-offs across system metrics, such as performance, data protection and power consumption. It also allows an automated optimization system to properly balance the cost of data migration with its expected benefits.

This work develops a prototype storage provisioning tool that uses an administrator-specified utility function to generate cost-effective storage configurations. The tool is then used to provide examples of how utility can be used to balance competing objectives (e.g., performance and data protection) and to provide guidance in the presence of external constraints. A framework for using utility to evaluate data migration is also developed. This framework balances data migration costs (decreases to current system metrics) against the potential benefits by discounting future expected utility. Experiments show that, by looking at utility over time, it is possible to choose the migration speed as well as weigh alternate optimization choices to provide the proper balance of current and future levels of service.

Acknowledgements

In the ten years that I spent in graduate school at CMU, I had the pleasure of collaborating with and learning from many very bright people. My advisor, Greg Ganger, has taught me much about what it means to do computer systems research — to ask the interesting questions, to dig for the answers, and to write about them in a compelling way. I am very grateful that, during this process, he permitted me the latitude to find a research topic that suited my interests. I would also like to thank the other members of my thesis committee. Their feedback has greatly improved both the outcome and presentation of my work.

There are many people that contribute to the success of a research group and its individual members. For my entire time as a graduate student, Karen Lindenfelser has always been there to support the students, faculty, and other staff members. She has helped me on more occasions than I can count. Whether I needed help with some important deadline or just needed to schedule a conference room, she was always willing to help. Joan Digney, another long-time PDL staff member, has ensured that the group's work has a professional, polished look. I have given her many rough block diagrams and sketches that she (seemingly effortlessly) has turned into professional-looking pictures.[1]

My family has also contributed greatly to my achievements with their support and encouragement. My wife, Corley, has shown a great deal of patience with the amount of time I have spent in school. We started graduate school at the same time, though she received her degree [Strunk, 2003] long before I. I am grateful to have someone with whom I could share my successes and that would understand the inevitable complaints. My parents set a foundation for life-long learning by fostering my curiosity and ensuring that I always had the resources to pursue my interests. It was their foresight in purchasing our family's first computer (in the mid-1980s) that sparked my interest. With my parents help and encouragement, it was on that computer that I wrote my first programs.

[1] See Figure 1.3 for an example. I have omitted my original sketch to save the embarrassment.

I thank the members and companies of the PDL Consortium (including APC, Cisco, EMC, Google, Hewlett-Packard, Hitachi, IBM, Intel, LSI, Microsoft, NetApp, Oracle, Seagate, and Symantec) for their interest, insights, feedback, and support. Experiments were enabled by hardware donations from Intel, APC, and NetApp. This material is based on research sponsored in part by the National Science Foundation, via grant #CNS-0326453, and by the Army Research Office, under agreement number DAAD19–02–1–0389. Experiments were run using the Condor [Litzkow et al., 1988] workload management system.

Contents

Abstract i

Acknowledgements ii

1 Introduction 1

1.1 Thesis statement . . . 3

1.1.1 Distributed storage system . . . 4

1.1.2 Guiding design and optimization . . . 4

1.1.3 Expressing objectives via utility . . . 6

1.2 Tool overview . . . 7

1.3 Overview of rest of document . . . 7

2 Background 10

2.1 General related work . . . 11

2.1.1 Specifying objectives . . . 11

2.1.2 Previous tools and approaches . . . 12

2.2 Storage architecture . . . 14

2.2.1 Ursa Minor architecture . . . 15

2.2.2 Modeling Ursa Minor . . . 19

3 Storage system models 21

3.1 Availability . . . 21

3.1.1 Detailed model . . . 23

3.1.2 Binomial model . . . 24

3.1.3 Improvements and related models . . . 25

3.2 Capacity . . . 26
3.3 Cost and power . . . 27
3.3.1 Improvements and related models . . . 28
3.4 Management complexity . . . 28
3.4.1 Improvements and related models . . . 29
3.5 Performance . . . 30
3.5.1 Queueing model . . . 31
3.5.2 Improvements and related models . . . 38
3.6 Reliability . . . 39
3.6.1 Markov model . . . 39
3.6.2 Improvements and related models . . . 42
3.7 Other models . . . 42
3.7.1 Physical space . . . 43
3.7.2 Robustness . . . 43
3.7.3 Similarity . . . 43
3.8 Summary of models . . . 44
3.8.1 Model performance . . . 44

4 Utility **48**
4.1 Overview of utility . . . 48
4.2 Utility in computer systems . . . 50
4.3 Cost-based utility . . . 50
4.3.1 Examples . . . 52
4.4 Priority-based utility . . . 56
4.5 Utility with constraints . . . 58
4.6 Utility elicitation . . . 59

5 Provisioning solver **60**
5.1 Exhaustive solver . . . 61
5.2 Random solver . . . 62
5.2.1 Generating configurations . . . 63
5.3 Greedy solver . . . 64
5.4 Genetic solver . . . 65

5.4.1 Configuration representation 66
5.4.2 Fitness function 67
5.4.3 Selection function 68
5.4.4 Crossover 69
5.4.5 Mutation 69
5.4.6 Parameter tuning and stopping criteria 70
5.5 Comparison of solvers 73
5.5.1 Genetic solver performance 78
5.6 Overview of other potential optimization techniques 79
5.7 Summary 79

6 Provisioning and initial configuration 80
6.1 Trade-offs across metrics 81
6.2 Storage on a budget 83
6.3 Effects of hardware cost 85
6.4 Effects of model error 85

7 Migration solver 88
7.1 Trade-offs involved in migration 89
7.2 Valuing a migration plan 89
7.2.1 Comparison against the current configuration 92
7.3 Modeling migration 92
7.4 Searching for good plans 96
7.5 Migration summary 99

8 Automatic adaptation 101
8.1 Effect of discount rate 101
8.2 Incremental provisioning 103
8.3 Repair 107
8.4 Time to upgrade? 108

9 Conclusions 111
9.1 Contributions 111
9.2 Important future work 112

A Description of modeled components **114**

Bibliography **116**

List of Tables

2.1 General effects of encoding parameters on system metrics 18
2.2 Main components and their attributes 20

3.1 Storage metrics provided by system models 22
3.2 Detailed availability model example 24
3.3 Asymmetry model example 29
3.4 Summary of queueing model 37

6.1 Effect of workload importance 83
6.2 Designing for limited budgets 84
6.3 Price affects the optimal configuration 86
6.4 Effect of model error 87

A.1 Clients used in experiments 114
A.2 Storage nodes used in experiments 115
A.3 Workloads used in experiments 115

List of Figures

1.1 Spectrum of expressiveness 3
1.2 Matching data distribution to workload 5
1.3 Utility-based provisioning loop 8

3.1 Example Markov chain for reliability model 41
3.2 System model performance 45
3.2 System model performance (cont.) 46
3.3 Total time to calculate system metrics 47

5.1 Sensitivity to mutation probability 72
5.2 Sensitivity to crossover probability 73
5.3 Sensitivity to population size 74
5.4 Finding rare solutions 76
5.5 Convergence of the genetic solver 77
5.6 Genetic solver performance 78

8.1 Migration using four different discount rates 102
8.2 Migration plan for incremental provisioning 106
8.3 Comparison of two possible orderings for repair 109

Chapter 1

Introduction

Today's storage system administrators manage a large amount of data for a diverse set of users and applications. Each distinct set of users has different objectives for their data, such as performance, reliability, and availability specifications, and each application responds differently to a particular level of service.

In the current state of the art, system administrators are forced to manually decide how data should be stored. This generally takes the form of creating different pools of storage in an attempt to match workloads and datasets to their requirements, while not spending exorbitant amounts of time treating each dataset and workload individually. For instance, the administrator may create some volumes using expensive, fast disks for performance sensitive applications, while putting user home directories onto slower, yet reliable, storage. Temporary data may be assigned to a striped volume with no redundancy to maximize performance while keeping the cost low. Storage pools work reasonably well when there are relatively few data volumes, users, and applications, but as administrators are required to handle more, the complex interactions across workloads cause the difficulty of creating good solutions to grow quickly. Adding datasets and workloads to existing pools of storage can unintentionally degrade the performance of the existing workloads. Further, as new datasets are added, their objectives are unlikely to exactly align with the existing storage classes, potentially causing the administrator to add additional storage pools that must also be managed.

The objective is to have a customized pool of storage for each user/application that provides exactly the levels of service that are desired, taking into account costs, resource limitations, and relative importance. Additionally, the administrator would like to be able to treat each pool independently (e.g., not think about the e-mail system while configuring the customer database), yet

have a unified infrastructure for system provisioning, configuration, monitoring, and maintenance. It is also important to note that the administrator is ultimately interested in the level of service, not the mechanics of the system's configuration.

In traditional systems, the administrator is forced to interact directly with the storage system's mechanisms. For instance, the administrator must configure groups of disks into RAID volumes, choosing correct values for parameters such as the RAID level and stripe size. Setting these parameters is a means to an end — the administrator is actually attempting to create a configuration that has the proper balance of performance, availability, reliability, and cost (resource consumption). Raising the interface to specifying policies or desires as opposed to the mechanisms used to implement the policy is an important way of managing system complexity. It removes the specifics of the underlying hardware from consideration by the administrator. This requires a new way for the administrator to specify his desires. It also requires the system to automatically map those requirements to a configuration that meets those desires. It requires a *self-tuning storage system*.

To ease the burden on system administrators, storage systems must become self-tuning. A self-tuning system is one that has the ability to shift resources among users and applications dynamically, based on the current state of the system and applied load. This dynamic allocation of resources must take into account system objectives and their relative importance. It needs to automatically respond to problems, such as failures, and take action to protect data until the problems can be repaired. In general, system administrators appear willing to let the storage system take on some of these tuning tasks. Telford et al. [2003] studied database administrators and their acceptance of automatic tuning. Broadly, they found that database administrators welcomed suggested tuning actions, and the part-time administrators were the most receptive, including allowing the system to make automatic adjustments.

To combat the complexity and expertise required to properly set the storage mechanisms, researchers have proposed using goal-based specifications [Anderson et al., 2003; Borowsky et al., 1997; Gelb, 1989; Wilkes, 2001]. These specifications free the administrator from the details of setting individual mechanisms, instead, allowing him to specify the desired levels of system metrics for each dataset and workload. Automated tools [Alvarez et al., 2001b; Anderson et al., 2002, 2005] could then be used to design and configure storage systems that meet these requirements.

While goal-based specifications raised the level of expressiveness, they still lacked the ability to express trade-offs between the many factors that impact storage costs and benefits [Kephart and Walsh, 2004]. Goal-based specification assumes the administrator can determine the exact level of each system metric that produces the proper balance of cost and benefit to provide the most cost-

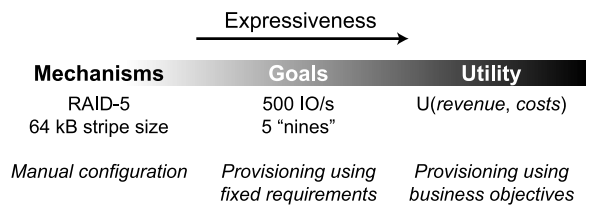


Figure 1.1: Spectrum of expressiveness – Moving from mechanism-based specification to goal-based specification allowed the creation of tools for provisioning storage systems to meet fixed requirements. Moving from goal-based to utility-based specification allows tools to design storage systems that balance their capabilities against the costs of providing a particular level of service. This allows the systems to better match the cost and benefit structure of an organization.

effective storage configuration. Clearly, this is a difficult, if not impossible, task for the administrator since costs are influenced by the mechanisms and hardware that goals try to insulate him from. To combat this, a method of specification that can encapsulate the trade-offs is necessary.

Using utility functions, a system administrator can express the costs and benefits of potential storage configurations in a manner that the design, configuration, and tuning of storage could be automated. Figure 1.1 shows the progression of expressiveness from mechanisms, through goals, to utility functions. Utility functions add the ability to make trade-offs across system metrics to create cost-effective storage configurations.

The remainder of this chapter presents and explains the thesis statement. It also provides an overview of the rest of the document, outlining the experiments and results that demonstrate the validity of the thesis statement.

1.1 Thesis statement

The focus of this work centers on the following statement:

> *Utility functions are a promising method for guiding the design and optimization of a distributed, self-tuning storage system.*

In the subsequent sections, the details of this statement will be broken into parts and explained, framing the problem scope and the actions necessary to demonstrate the overall statement's validity.

1.1.1 Distributed storage system

The architecture used as a context for experimentation is that of CMU's Self-∗ Storage prototype, Ursa Minor [Abd-El-Malek et al., 2005]. Ursa Minor is a prototype versatile, cluster-based storage system, composed of *clients*, *storage nodes*, and a *metadata service*. Clients contact the metadata service to determine the location of data objects and to obtain access permission. They are then able to directly communicate with the system's storage nodes to read and write data. This type of architecture provides a large amount of incremental scalability through the addition of storage nodes. With these scalability benefits come an associated increase in the number of configuration options. Instead of creating fixed volumes of storage, like would be found in traditional, monolithic arrays, the Ursa Minor architecture supports arbitrary m-of-n data encoding schemes for objects. This allows each data object to have a customized level of fault tolerance and choice of storage nodes, providing the ability to customize the properties of the storage system to the needs of applications and datasets.

Figure 1.2 demonstrates the gains that are possible via customization and the breadth of trade-offs that are possible. The graph compares four different workloads using data distributions that are customized to each, as well as an additional one-size-fits-all data distribution. By creating an appropriate data distribution, a dataset such as "Scientific" can be customized for high throughput at the cost of availability and reliability, or an "OLTP" dataset can be tuned for high availability and small I/O requests. These very apparent benefits from customization show that there are significant tuning options that have a measurable impact on the service provided. This makes the Ursa Minor architecture a good platform to use for investigating the use of utility to guide tuning.

1.1.2 Guiding design and optimization

A self-tuning storage system is able to automatically make adjustments to its configuration without the direct intervention of a system administrator. With such a system, it must be possible to both design a good initial configuration and to provide a framework that allows automatic optimizations to proceed in an intended manner. These two requirements are examined separately.

Initial provisioning and configuration is the ability to begin with a set of requirements, goals, and constraints and produce a storage system that meets them in a cost-effective manner. Framing this in a cluster-based architecture, the datasets, clients, and I/O workloads are defined, and the task is to instantiate some number of storage nodes and assign the datasets to them. A good provision-

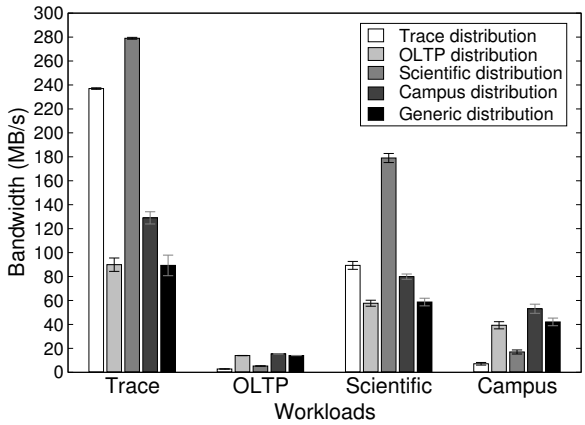


Figure 1.2: Matching data distribution to workload – This graph shows the benefit of customizing a storage configuration to the needs of the application. It shows the performance of four workloads run on Ursa Minor as a function of the data distribution. For each workload, five distributions were evaluated: the best distribution for each of the four workloads and a good "middle of the road" choice for the collection of workloads. Although the "Scientific" data distribution provided better performance for the "Trace" workload than the "Trace" distribution, and the "Campus" data distribution provided better performance for the "OLTP" workload than the "OLTP" distribution, these distributions failed to meet the respective workloads' reliability requirements. These numbers are the average of 10 trials, and the standard deviations are shown as error bars. This graph originally appeared in Abd-El-Malek et al. [2005].

ing solution produces a set of hardware (storage nodes) and assignments (data distributions) that maximize (or minimize) some metric.

Once a system has been configured and put into service, changes to the resources, objectives, or workloads can cause the system configuration to become inadequate. This could be caused by changes in system resources, such as the addition or removal of resources (e.g., from storage nodes failures, repair, or system upgrades). Additionally, system objectives can change over the lifetime of a dataset. For instance, a dataset could be removed from active service and instead be maintained for archival purposes. Workloads can also change over time, changing intensity or I/O pattern as the popularity or usage pattern of the associated application changes. These changes may require adjustments to the storage system's configuration, and a self-tuning system should be able to make these changes without the direct intervention of a system administrator. Critical to this

task is determining potential new configurations that better meet the (new) system objectives and controlling the migration of data from the old to the new configuration. The choice of whether to optimize, and how fast, is frequently a trade-off between the current level of service and the level of service at some future time. A technique that is able to control a self-tuning system should be able to automatically navigate these trade-offs.

1.1.3 Expressing objectives via utility

Utility is a value that represents the desirability of a particular state or outcome. This concept is common in both economics [Browning and Zupan, 1996] (to explain consumer preferences) and in decision theory [Keeney and Raiffa, 1993] (as a method for weighing alternatives). The main feature used here is its ability to collapse multiple objectives (e.g., performance and reliability) into a single axis that can be used to compare alternatives.

To use utility to guide storage tuning, it is necessary to have a utility function that is able to evaluate a potential storage configuration and produce a single value (its utility) that can be compared numerically against other candidate configurations. The optimal configuration is the one with the highest utility value. The utility value for a configuration should be influenced by the system metrics that are important to the administrator. For example, configurations with high performance would have higher utility values than those with low performance; likewise for availability and reliability.

Examining system metrics in isolation, one could use the actual metric (or its negation or reciprocal) as the utility value. For example, setting $Utility = Bandwidth$ would cause the provisioning system to prefer systems with high bandwidth over those with low bandwidth. The goal of utility, however, is to combine all relevant system metrics into a single framework. The various metrics cannot simply be summed; they must be combined in a manner that captures their relative importance. The different metrics must be normalized or scaled relative to each other. Experience suggests that the easiest method for normalizing these different metrics is via a common scale that has meaning for each metric. One such scale is money (e.g., dollars). Since each storage metric has an effect on the service provided, it affects an organization's business. This business impact can be expressed in dollars. For example, performance (throughput) affects the number of orders per second that an e-commerce site can handle, and a loss of availability causes lost business and decreased productivity. By expressing each source of utility (e.g., performance, data protection, and system cost) in dollars, they can be combined easily. It is important to note, however, that the "sources" or components of utility do not necessarily have a one-to-one correspondence with system metrics. For example,

the revenue of a business may be a source of utility, but that revenue may be a function of both performance (throughput) and availability.

1.2 Tool overview

As a part of evaluating the use of utility for storage provisioning and tuning, a pair of tools were created. The first, an automated provisioning tool, uses utility to create an initial storage system configuration. The second, a dynamic tuning tool, uses utility to evaluate different tuning alternatives. The two tools share most of their architecture and source code but differ in the parameters they configure and how they generate solutions. This section describes the high-level architecture of these tools, placing the following chapters in context.

Figure 1.3 presents a graphical overview of the architecture of the provisioning tool. It tool has three main components. The first component is a set of system models that analyze a prospective system configuration to produce a collection of high-level system metrics. The models use low-level configuration information such as the m, n, and locations of each dataset, and they produce system metrics such as the number of IO/s each workload would complete. The second component is the utility function, which calculates the utility of the configuration using the high-level system metrics. The utility function encapsulates the system administrator's objectives for the system, workloads, and datasets. The third component of the tuning loop is the solver. The solver uses the utility values to guide the optimization, producing new configurations based on the results of previous analyses.

The dynamic tuning tool leverages the provisioning tool to evaluate potential new storage system configurations. It also uses this framework to evaluate the expected level of utility during reconfiguration and to optimize the order and speed of the optimization steps.

The system models, and the metrics that they produce, are discussed in detail in Chapter 3. Utility and utility functions are described in Chapter 4. The two different solvers are presented separately. Chapter 5 discusses the solver used for utility-based storage provisioning, and Chapter 7 describes the solver used for data migration.

1.3 Overview of rest of document

Following the criteria of Section 1.1.2, the document is organized around the two main tasks of storage provisioning and dynamic tuning. First, Chapter 2 provides background on previous provi-

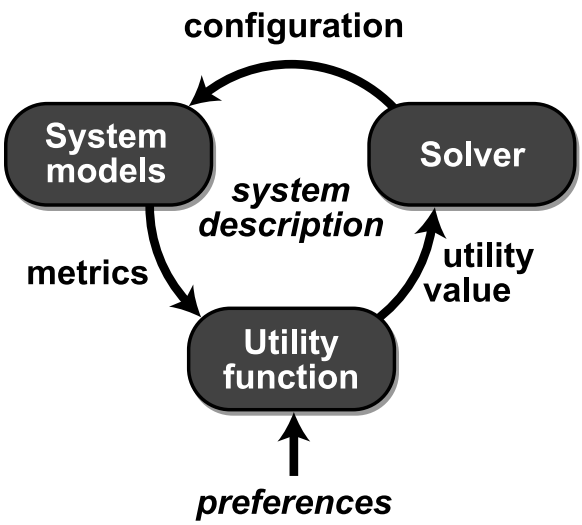


Figure 1.3: Utility-based provisioning loop – The solver produces candidate system configurations. The system models annotate the configurations with system, workload, and dataset metrics. The utility function uses the administrator's preferences to rank the annotated configurations by assigning a utility value to each.

sioning and automated tuning approaches. It also discusses the Ursa Minor architecture and how it is modeled in this work. Chapter 3 presents various system models used to produce the system metrics that form the building blocks for utility functions. Chapter 4 describes utility in more detail, highlighting other uses of utility in computer systems and discussing how an administrator might create utility functions for his environment. Chapter 5 describes the implementation of several solvers that are used to generate and compare storage configurations for automated provisioning.

The building blocks of models/metrics, utility functions, and the solvers are used to examine storage provisioning in Chapter 6. This chapter highlights some of the benefits of using utility for storage provisioning. It shows how cost-effective solutions can be created using utility functions and how these solutions may defy the conventional rules of thumb for manual provisioning.

Chapter 7 describes how utility can be used to decide whether new configurations should be implemented and how fast data should be migrated. It discusses examining utility over time to determine the proper trade-off of current versus future utility and how this can be used to control data migration. Chapter 8 provides some examples of dynamic tuning using utility. It highlights the impact of the system administrator's ROI time horizon on tuning decisions and shows how different

system scenarios are optimized. Chapter 9 presents the conclusions and discusses future work to help move these techniques toward commercial viability.

Chapter 2

Background

Managing storage is expensive. In 2000, Gartner Consulting estimated that 55% of the TCO for a server could be attributed to the storage [Gartner Consulting, 2000]. In the same report, they claim that managing local storage costs just under $23,000 per TB-yr. It also estimated that management of LAN-based storage required, on average, 10 hr per GB-yr. In a 2003 interview [Gray, 2003], Jim Gray provided some estimates for the cost of managing storage, saying that his "Wall Street friends" claim it costs them $300,000 per TB-yr to manage storage and that they have over one administrator per 10 TB. He also estimated that Google and the Internet Archive have one administrator per 100 TB. Even though these estimates vary by several orders of magnitude, all indications are that the cost of managing storage is high.

One reason for the expense of managing storage is how vital it has become to the proper functioning of the enterprise. A 1996 survey of the cost of downtime [Contingency Planning Research, 1996] estimated that the financial impact to companies ranged from $14,000 to over $6.4 M per hour. They also cite data from the University of Texas that indicates 94% of businesses that suffer a catastrophic data loss go out of business within two years. In an updated study from 2001 [Eagle Rock Alliance, Ltd., 2001], over 50% of the companies surveyed responded that downtime costs them over $50,000 per hour, and 8% of respondents placed their cost at over $1 M per hour. With the high cost of managing storage and the penalties for doing it poorly, it is no surprise that there is a push toward improving manageability.

This chapter begins with a high-level survey of storage management research. It then describes the architecture and system that provides the context for this work.

2.1 General related work

While much of the related work is discussed as the topics are presented, this section presents some of the work that has set the stage for the investigation presented here. It discusses previous approaches to specifying system objectives as well as tools and approaches that have been used to both provision and tune storage systems.

2.1.1 Specifying objectives

To allow a system to effectively provide automated configuration or adaptation, it must be possible for the administrator to communicate his desires to the storage system. These desires provide a target, or goal, for the system to achieve. Gelb [1989] notes that, in the 1970's and 80's, the increase in customer demands on storage brought about the need for customers to be able to logically express their storage needs. This need was the main drive behind IBM's System-Managed Storage effort. SMS was an attempt to raise the level of specification of a storage system above the low-level configuration settings that implement a storage service to one that describes what the service should provide. Borowsky et al. [1997] and Shriver [1999] defined the Attribute Managed Storage concept that has guided a large amount of work from HP Labs. It framed storage specifications as a language of constraints that a storage service must meet in order to satisfy the end user or application.

Specifying storage objectives can also be viewed as a direct translation of work on specifying quality of service (QoS) specifications that have been an active research area in systems and networks. For example, QML [Frølund and Koistinen, 1998] attempted to create a QoS specification language for use in designing software systems. Wilkes [2001] developed a language, called Rome, specifically for communicating storage designs and goals.

Attempts to separate the specification of storage objectives from the mechanisms that implement them have not solved the problem of administrators being able to specify their objectives effectively. Determining a suitable set of objectives is difficult in many cases, because even the high-level metrics of a storage system are still far from the true metrics that interest the administrator and end users. Anderson et al. [2003] presents a set of ideas that could help ease management, including not only application-level specifications, but also using analogies to allow administrators to make specifications based on current systems, in a relative manner (e.g., half as large or twice as fast).

Another difficulty is the separation of costs from benefits. Determining a proper set of "requirements" requires considering the costs involved in meeting them. There are few environments where the costs are either unimportant or so small as to be negligible. In situations where costs need to be

considered, asking for a fixed set of service requirements is not an appropriate solution because the requirements are shaped by the costs necessary to meet them. This is one of the benefits of moving toward a method of specifying storage objectives based on utility. Recently, there has been work on designing cost-effective disaster recovery solutions, trading off solution costs with expected penalties for data loss and downtime [Gaonkar et al., 2006; Keeton et al., 2006, 2004]. This work has effectively used utility to trade off the costs of data protection mechanisms against the penalties when data is lost, creating minimum (overall) cost solutions for disaster protection. These results lend support to the notion of using business costs (and benefits) as the basis for evaluating storage solutions.

2.1.2 Previous tools and approaches

Many provisioning and tuning approaches have been concerned only with performance. This is a natural place to start because more performance is generally better than less, no matter the current level. However, there is an unmentioned assumption in much of the work that the performance optimizations will not affect other metrics, such as data protection. Choosing just a few examples from an large collection, Wolf [1989] describes the Placement Optimization Program (POP) which is designed to solve the "File Assignment Problem." POP used a two-stage approach, first determining a target access rate for the disks, then assigning files to those disks to best match the target I/O rates. Weikum et al. [1990] describe a system called FIVE that spreads data across a number of disks in a flexible manner, allowing the system to balance I/Os with the objective of improving performance. Their algorithms were designed to both balance I/O load across disks and maintain good sequential performance for files. The AutoRAID system, described by Wilkes et al. [1996], dynamically moves data between mirrored and RAID-5 storage based on the amount of system free space and the recent access history.

Rule-of-thumb approaches to storage allocation are also common. For example, Loaiza [2000] presents a scheme called "Stripe And Mirror Everything" or "S.A.M.E." for configuring Oracle databases. Instead of attempting to assign storage resources to each database volume, he advocates spreading data evenly in an attempt to provide a consistent (and hopefully high) level of performance to all datasets. For this, he proposed a two stage approach. First, use mirroring for all storage to provide adequate data protection. Second, stripe all data across the available hardware to ensure the load is spread evenly. Another example is the creation of "storage classes" [St.Pierre, 2007] whereby the administrator creates broad categories for data such as "mission critical" or "archival"

and assigns datasets to pools of storage that are (hopefully) provisioned to match the objectives of these categories. One problem with these approaches is that the intended level of service may differ significantly from the actual, and there is little feedback to recognize and correct the problem. For example, a "high performance" storage class may initially meet expectations, but as more datasets and high-intensity workloads are assigned to the storage pool, its performance will inevitably degrade.

One method of attempting to automate the rules-of-thumb-type approach to storage tuning is via Event-Condition-Action (ECA) rules. These systems are based on a set of rules that are triggered by some system state or event. The rules then apply a particular action to adapt to current conditions. Unfortunately, ECA systems tend to be difficult to implement, because the potential for rule conflicts grows with the number of rules. Additionally, while each rule may be simple, the complex behavior produced by the collection may not be as intended. Polus [Uttamchandani et al., 2004] was an attempt to apply an ECA model to storage optimization, using a learning framework to assist in creating the rules that guide the tuning.

Automating storage provisioning allows creating designs that are specifically tailored to the objectives of the applications and datasets. For example, Minerva [Alvarez et al., 2001b] divided the provisioning task into two phases. The first phase determined the proper encoding for the data (i.e., whether it should be placed on a RAID-10 or a RAID-5 volume). The second phase packed the datasets into the appropriate volume type, accounting for capacity and other resource limitations. As a response to limitations caused by the two-phase process of Minerva, Ergastulum [Anderson et al., 2001b] was created. It used a series of bin-packing heuristics to place datasets onto storage volumes. The authors showed that it generally produced better storage designs (using less hardware) and accomplished this much quicker than Minerva. Ergastulum served as the solver for Hippodrome [Anderson et al., 2002], which iteratively refined storage designs via a tuning loop that instantiated and measured successive designs. Ergastulum was later renamed the Disk Array Designer and discussed in detail by Anderson et al. [2005].

The most closely related work investigated automatically creating data protection configurations to minimize the expected costs to an organization in terms of both the cost of failure (in outage and data loss penalties) and the cost of the data protection system itself. Keeton and Wilkes [2002] developed a detailed vocabulary for expressing data protection needs and characteristics. This vocabulary was then used for "what-if"-type exploration of the design space [Keeton and Merchant, 2004] and as the basis of an automated data protection design tool [Keeton et al., 2004]. These techniques were further extended to automate failure recovery [Keeton et al., 2006] and to work in

multi-application environments [Gaonkar et al., 2006]. This body of work used utility, defined as the expected total cost of data protection (failures and the protection thereof) as the objective for optimization. This utility function incorporated the administrator's objectives via the use of two penalty rates. The "outage penalty rate" was defined as a cost penalty that accumulates during the time that a dataset is unavailable, and the "loss penalty rate" was a cost penalty based on the amount of recent updates that are lost during a failure. Both were expressed as a cost per unit of time (e.g., dollars per hour).

While this dissertation concentrates on a different area of storage provisioning (provisioning and tuning of the primary storage system), it extends this previous work in several ways. The administrator's preferences can be specified in a more general way, via a set of functions. This method for specifying utility allows more flexibility in representing the cost structure of the particular target environment. For example, it allows storage metrics to be combined, such as to express revenue as a function of both performance and availability. It also directly permits functions of storage system metrics, such as the time required to scan through a dataset being proportional to the reciprocal of bandwidth. See Section 4.3.1 for specific examples. This same utility framework can be used to add external constraints, such as a limited budget, to the optimization. While previous work could potentially incorporate this as additional constraints in the optimization, there is value to keeping all of the administrator's preferences within the same framework. The move to supporting more general expressions of utility required a different optimization technique than the Mixed Integer Programming used by Keeton et al. [2004]. Section 5.4 presents a solver based on a genetic algorithm. Gaonkar et al. [2006] also found it useful to use a genetic algorithm for designing for multiple applications, and Keeton et al. [2006] used one for ordering recovery operations.

The tuning portion of this dissertation is most closely related to the use of utility for scheduling computation in the face of deadlines. This previous work constructed utility functions that decayed as tasks were delayed [Irwin et al., 2004], and used utility to express the desire to have a group of computational jobs complete together [AuYoung et al., 2006]. While both of these had a time component to their expression of utility, this dissertation introduces the notion of using a discount rate to balance the current and future benefits and costs of data migration.

2.2 Storage architecture

The desire for storage scalability has pushed researchers to look toward storage clusters. Storage clusters are composed of a (potentially) large number of individual storage nodes or servers that act

together to implement a larger storage service. The original motivation for storage clusters such as Petal [Lee and Thekkath, 1996], AFS [Howard et al., 1988], xFS [Anderson et al., 1996], or NASD [Gibson et al., 1998] was the large amount of scalability that could be gained by allowing servers to cooperate. More recently, there has been an interest in "brick-based" storage which differs mainly in the characteristics of the individual node and the dynamic nature of the system. Brick-based storage systems, such as FAB [Saito et al., 2004], Kybos [Wong et al., 2005] and Ursa Minor [Abd-El-Malek et al., 2005], tend to use larger quantities of smaller storage nodes (bricks). In some cases, such as FAB and Ursa Minor, they place an emphasis on using commodity-class components to reduce the cost of the system while leveraging the large number of components to provide redundancy as well as array-like performance and data protection. The notion of cluster-based storage is also appearing in commercial products. For example, Network Appliance's GX [Eisler et al., 2007] system can be traced back to the AFS project. The Google File System [Ghemawat et al., 2003] is a massive storage system, with some similarity to NASD, used by Google to store search and user data. There are other examples as well.

One of the more interesting aspects of cluster-based storage is the large number of tuning possibilities that it presents. Traditional arrays are generally limited in the data distributions that they allow (e.g., limited data placement and a few RAID levels), but cluster-based approaches tend to provide more varied options, including arbitrary m-of-n erasure codes for data protection. This increase in the number of configuration options has the potential to increase the difficulty of management, making the architecture a good choice for study.

2.2.1 Ursa Minor architecture

Ursa Minor [Abd-El-Malek et al., 2005] is a prototype versatile, cluster-based storage system from the Self-$*$ Storage [Ganger et al., 2003] project. Self-$*$ storage systems are designed to be self-managing, and Ursa Minor is the first step toward that goal. The prototype incorporates much of the versatility and mechanisms that will be useful for a system that automatically adapts to changing environments. The system presents an object-based interface, similar to that of the NASD [Gibson et al., 1998] project. It allows clients to directly access data objects from storage nodes after receiving location information and permission from a metadata service.

The storage nodes are envisioned as storage "bricks." They are typically realized with PC-class hardware, processing, and memory and from one to one dozen disks. The notion of a storage "brick" arises because they are small, self-contained entities that can be assembled to form a larger

storage system. Storage nodes internally handle data allocation and placement, presenting a block-addressed object interface for external access. Each block is addressed by its object-id and a block number. The blocks hold a variable amount of data and must be read and written as a unit.

The metadata service controls access and maintains location and encoding information for each object. Each object's metadata contains one or more slice descriptors that describe the data distribution for an extent of the object. A data distribution is composed of a data encoding, describing how a data block is divided into fragments, and location information, which lists the storage nodes that hold fragments for the data blocks in the slice (extent).

Clients Clients access data by first contacting the metadata service and retrieving the appropriate slice descriptors for the object. Once they possess the metadata for an object, clients may directly interact with the storage nodes to read and write. For a write operation, the client breaks the data block into fragments as specified in the slice descriptor's encoding information. Then, it sends the fragments to the storage nodes listed in the location information. For reads, data fragments are retrieved directly from storage nodes and decoded by clients to recover the original data block.

Read/write protocol The PASIS protocol [Goodson et al., 2004] is used for data transfers between clients and storage nodes. The protocol provides consistent data writes across multiple storage nodes in the presence of concurrent operations from multiple clients. The PASIS read/write protocol is an optimistic protocol that resolves conflicts caused by concurrent writers when data is subsequently read. This eliminates much of the overhead to providing data consistency, but it relies on storage nodes that are able to version the data.

The PASIS protocol is actually a family of protocols that can be adapted for different system environments. The protocol can be configured for either synchronous or asynchronous operation. The former allows more efficient operation, because the response time for requests is bounded, while the latter has the benefit that its correctness does not rely on storage nodes to respond within a strict time period. Asynchronous operation prevents overloaded servers and network partitions from interfering with the correct operation of the system, but this resiliency typically comes at the cost of increased communication for each operation. The protocol also allows a choice of the quantity of storage node failures to tolerate as well as the type of those failures. The protocol can tolerate up to t total failures, and up to b of those may be Byzantine. The remaining failures, $t-b$, must be benign (e.g., crash-only) failures. The modeling and evaluation in this dissertation is concerned only with a synchronous timing model and crash failures (i.e., $b = 0$).

The encodings for data blocks use erasure codes, wherein a data block is divided into n fragments when it is written, and any m of those fragments can be used to recreate the original data block. The encoding parameters, n and m, are chosen to coincide with the read and write quorums of the PASIS protocol. Bounds on the number of fragments that must be read or written for a given level of fault tolerance are described in Goodson et al. [2003]. For the specific protocol configuration used here, data is written to n storage nodes, where $n = m + t$, and m is restricted to values greater than or equal to one. Using values for m that are larger than one allow more space efficient data encodings at the cost of communicating with more storage nodes and additional encode/decode computation for each I/O operation.

The design of Ursa Minor contained the notion of specifying more storage nodes (data locations) for a given slice than the minimum of n. With a larger list, blocks could be stored on different subsets of size n, most likely by storing fragments on n consecutive nodes in this list of l storage nodes, rotating which storage node receives the first fragment based on the block number within the slice. For example, an encoding may be specified as 2-of-3 declustered across 4 with the list of storage nodes containing storage nodes sn1 through sn4. The first block would be stored on nodes sn1, sn2, and sn3 using a 2-of-3 scheme. The second block would be on sn2, sn3, and sn4. The third block would reside on sn3, sn4, and sn1. This rotation across a larger set of nodes allows the number of storage nodes for a particular block to be limited, helping I/O latency, while allowing a larger number of nodes to serve the object as a whole, helping bandwidth.

To simplify the modeling effort, the model used in this work differs slightly from the one used in Ursa Minor. While Ursa Minor rotates blocks across the set of l storage nodes, the modeling for this work assumes that each block is stored on a random subset. This should have a minimal impact on performance, but it does affect the availability and reliability calculations. By assuming an arbitrary, random subset of the l storage nodes, it is likely that any loss of $n - m$ of the storage nodes from the set of l would result in a loss, because for a large object, it is reasonable to expect that there is at least one block that shares any subset of size n of the l storage nodes. With a rotation model, this is not necessarily the case.

The choice of encoding parameters impact the access characteristics of the dataset. How these relationships are modeled is discussed in Chapter 3, but to provide some indication of the complexity of choosing a good data distribution, Table 2.1 shows how various workload and dataset metrics are affected by changes to these encoding parameters.

Table 2.1: General effects of encoding parameters on system metrics – This table shows the general effects on various system metrics caused by increasing the data encoding parameters, m, n, or l. The magnitude of these effects vary considerably and are difficult to quantify without detailed models. Making the proper encoding choice manually is very difficult because changing a single parameter affects nearly all the metrics. A system that is able to choose these parameters automatically must be able to make trade-offs across metrics.

Metric	$m \Uparrow$	$n \Uparrow$	$l \Uparrow$
Availability	$\Downarrow$	$\Uparrow$	$\Downarrow$
Reliability	$\Downarrow$	$\Uparrow$	$\Downarrow$
Capacity consumed	$\Downarrow$	$\Uparrow$	–
Read bandwidth	$\Downarrow$	$\Uparrow$	$\Uparrow$
Read latency	$\Uparrow$	–	$\Downarrow$
Write bandwidth	$\Uparrow$	$\Downarrow$	$\Uparrow$
Write latency	–	$\Uparrow$	$\Downarrow$

Storage nodes The storage nodes used for Ursa Minor are based on the storage server originally built for the Self-Securing Storage project [Strunk et al., 2000]. The benefit to using this as a starting point was its efficient versioning file system [Soules et al., 2003]. The storage server was used for the PASIS project. From there, it was modified for use in Ursa Minor. It presents an object-based interface to clients and internally manages allocation and data placement.

Data migration Ursa Minor supports the online migration and re-encoding of data. The migration is supervised by the metadata service, and it relies on a combination of the foreground workload and the migration coordinator to move the dataset from the old encoding and location to the new.

Migration is initiated when the metadata service installs *back-pointers* at the new data location. These back-pointers contain the necessary metadata to locate data in the old data distribution. Once the back-pointers are installed, all clients are redirected to the new dataset location. At this point, writes by the foreground workload use the new location and encodings, helping the data migration. Reads first attempt to retrieve data from the new location, but may retrieve a back-pointer instead of data if the requested block has not yet been migrated. The client can follow this back-pointer to retrieve the data from the old location.

The migration coordinator is responsible for moving the bulk of the data. It acts much like a

storage system client, reading data from the old location and writing to the new. By using special time stamps within the PASIS protocol, the actions of the coordinator are guaranteed to not overwrite data blocks already migrated by the foreground workload. Abd-El-Malek et al. [2005] provide the details of this time stamp handling.

2.2.2 Modeling Ursa Minor

This dissertation uses the architecture of Ursa Minor as a framework for evaluating the use of utility functions for guiding storage provisioning and tuning, but it does not attempt to exactly replicate the details and complexity of the prototype system. The modeling is designed to be complex enough for a thorough evaluation of utility while not devoting an exorbitant amount of effort to system modeling. Toward this end, the analysis neglects the metadata service and metadata operations, instead focusing solely on the storage system data path.[1] The model contains arbitrary numbers of clients and storage nodes, both of which may be composed of heterogeneous components. Datasets correspond to objects within the system, but to make the modeling more tractable, datasets are assumed to correspond to volumes (i.e., relatively few and of GBs in size) as opposed to files as in the Ursa Minor prototype. Clients and storage nodes are assumed to be attached by a network capable of full port bandwidth between any two endpoints. Workloads are assumed to execute on a single client, accessing a single dataset. However, a client may have many workloads assigned to it, and a dataset may receive I/Os from many different workloads.

Each of these components, clients, workloads, datasets, and storage nodes are described by a set of attributes. For example, a storage node has a defined capacity, network bandwidth, and service rate for its disk. Several types of each component may be defined and used within the same storage system. Table 2.2 lists the attributes that are used to define each component.

The main parameters that are available for tuning are each dataset's m and n encoding parameters as well as the set of storage nodes on which it resides (the size of this set is the parameter, l). For provisioning, the system chooses a collection of, potentially heterogeneous, storage nodes to hold the datasets. It also chooses the encoding parameters for each. Additionally, it has the ability to throttle the foreground workloads to limit their resource usage.

For data migration, the system is given an initial configuration, and it determines a new configuration as well as the ordering of dataset migration and the speed of each of those migrations based

[1] Given the large size of the datasets (e.g., volumes) that are the focus of this work, the metadata service would not be expected to significantly affect system performance.

Table 2.2: Main components and their attributes – This table lists each of the main component types used in the system description for the provisioning tool. With each of the component types is the set of attributes that define their properties. Each instance of a component (e.g., each storage node) may have different values for these attributes, allowing the tool to evaluate heterogeneous configurations.

Client	
CPU delay	CPU time for data encode/decode (s)
Net bandwidth	Network streaming bandwidth (MB/s)
Net latency	Network propagation delay (s)
Dataset	
Size	Size of the dataset (MB)
Storage node	
AFR	Annual failure rate (%)
Availability	Fractional availability of the node
Capacity	Disk capacity (MB)
Cost	Purchase cost ($)
Disk bandwidth	Max streaming bandwidth (MB/s)
Disk latency	Initial positioning time (s)
Net bandwidth	Network streaming bandwidth (MB/s)
Net latency	Network propagation delay (s)
Power	Power consumption (W)
Workload	
I/O size	Avg. request size (kB)
MP level	Multi-programming level for closed workload
Think time	Think time for closed workload (s)
Random frac	Fraction of non-sequential I/Os
Read frac	Fraction of I/Os that are reads

on utility. To control the speed of migration, the speed of the migration coordinator is controlled throughout each phase of the migration.

Chapter 3

Storage system models

A utility function specifies the utility value associated with given levels of relevant metrics, such as the availability, reliability, or performance of the datasets and workloads. To evaluate the utility of a potential system configuration, it must be possible to estimate the level of each of these metrics. This is the job of the storage system models.

Storage system *models* generate, for a given system configuration, a set of one or more *metrics*. The metrics produced by the system models form the building blocks for expressing storage objectives as utility. For example, to evaluate a throughput objective for a workload, there must be a performance model that produces a metric corresponding to the throughput of the workload. This chapter describes the system models that were implemented as a part of this work. Models can be added and removed easily, creating the ability to add new storage metrics or enhance existing ones. This, in turn, increases the expressiveness of utility for the administrator.

Table 3.1 provides an overview of the metrics that are used in future experiments. The table provides a brief summary of each metric as well as the model that is responsible for producing it. These metrics form the framework for expressing system objectives and creating utility functions.

3.1 Availability

Availability, $\mathrm{A}(t)$, is the probability that a dataset can be accessed at some particular time in the future. Taking the limit of this function as t goes to infinity yields the expected fraction of time

Table 3.1: Storage metrics provided by system models – This table lists the metrics that are added to candidate configurations by the current set of system models. The table is organized by the component to which the metric refers, and it lists the model that provides the metric.

Client	
CPU utilization (%)	Performance §3.5
Network utilization (%)	Performance §3.5
Dataset	
Annual failure rate (%)	Reliability §3.6
Capacity blowup from encoding	Capacity §3.2
Fractional availability (%)	Availability §3.1
Mean time to failure (hr)	Reliability §3.6
"Nines" of availability	Availability §3.1
Storage node	
Raw capacity consumed (MB)	Capacity §3.2
Capacity utilization (%)	Capacity §3.2
Disk utilization (%)	Performance §3.5
Network utilization (%)	Performance §3.5
Power consumed (W)	Cost and power §3.3
System-wide	
Total capacity consumed (MB)	Capacity §3.2
Capacity utilization (%)	Capacity §3.2
Total system cost ($)	Cost and power §3.3
System power consumed (W)	Cost and power §3.3
Management complexity	Management complexity §3.4
Workload	
Bandwidth (MB/s)	Performance §3.5
Throughput (IO/s)	Performance §3.5
Request latency (s)	Performance §3.5

that the dataset is available [Siewiorek and Swarz, 1982]. The availability model produces a metric based on this fractional availability number.

Two different models were created, each with a different blend of detail versus speed of evaluation. They both rely on the same input parameters (the fractional availability of individual storage nodes) and produce the same metrics (the fractional availability of each dataset), making them inter-

changeable. The difference between the two models is how they account for datasets that have more than one data distribution — a case that occurs during data re-encoding. The "detailed model" properly accounts for storage nodes that are part of both of the data distributions, while the "binomial model" assumes independence, potentially underestimating the availability. The binomial model also assumes all storage nodes involved in a data distribution have the same fractional availability while the detailed model does not. For both models, individual storage node failures are assumed to be independent.

Both models produce the same availability metrics. They tag each dataset with a fractional availability and the corresponding "number of nines." The fractional availability is a real between zero and one, inclusive. The number of nines, $\mathrm{Av9}(a)$, for a given fractional availability, a, is:

$$\mathrm{Av9}(a) = -\log_{10}(1-a)$$

As a concrete example, an availability of 0.999 would be 3 nines of availability.

3.1.1 Detailed model

The detailed model was created to accurately capture the dependencies across data distributions for datasets with multiple data distributions and also to handle data distributions with heterogeneous storage nodes.

The model examines and calculates the availability of each dataset individually. For each dataset, it iterates through each data distribution, noting the unique storage nodes used for that dataset. It then exhaustively examines all combinations of these storage nodes being either up or down. For example, a dataset may have two data distributions with the first being 1-of-2 on nodes "sn1" and "sn2," and the second being 1-of-2 on "sn2" and "sn3." In this case, the model would examine all possible states of the storage nodes ($2^3 = 8$ possibilities). If, for a given configuration, all datasets are available, then the dataset is available, and the likelihood of being in that state is added to the total probability for the dataset to be available. An individual distribution is available as long as no more than $n-m$ storage nodes are unavailable. In this example, there must be no more than one failed storage node in either data distribution since they both use a 1-of-2 encoding. Table 3.2 presents the eight possible states from this example and how the dataset availability is calculated.

An advantage of this availability model is that, if multiple data distributions for a dataset share storage nodes (in the above example, they shared sn2), the calculated availability will account for the overlap. It also has the advantage that each storage node is examined individually, allowing them

Table 3.2: Detailed availability model example – This shows an example availability calculation for a dataset with two data distributions. The first distribution, d1, is 1-of-2 on nodes sn1 and sn2, while the second, d2, is 1-of-2 on sn2 and sn3. The three storage nodes have individual availabilities of 0.9, 0.9, and 0.95.

sn1	sn2	sn3	d1	d2	Probability
down	down	down	down	down	–
down	down	up	down	up	–
down	up	down	up	up	0.0045
down	up	up	up	up	0.0855
up	down	down	up	down	–
up	down	up	up	up	0.0855
up	up	down	up	up	0.0405
up	up	up	up	up	0.7695
				Availability (sum):	0.9855

to have different component availabilities. In the example above, sn1 and sn2 had an availability of 0.9, while sn3 had an availability of 0.95.

The disadvantage to this model is that it is exponential in the number of storage nodes that contain fragments from a given dataset. In the above example, only three total storage nodes contained data fragments, so the number of combinations that had to be evaluated was small. A configuration where a dataset has two distributions, each with ten storage nodes, could have up to $2^{20} \approx 1$ M combinations. Modern storage arrays regularly incorporate up to fourteen disks in a single RAID group, so a configuration with ten storage nodes is not unexpected. At such a scale, this model is impractical, leading to another availability model.

3.1.2 Binomial model

After noting the poor scaling of the above availability model, a less detailed model was implemented. This new model scales better with problem size. It views each data distribution independently, and uses a single fractional availability for all storage nodes that are part of a given distribution. These simplifications allow the availability calculation to be viewed as a set of Bernoulli Trials, where the distribution is available as long as there are no more than $n - m$ failures. This leads

to the following expression:

$$Availability = \sum_{f=0}^{n-m} \binom{l}{l-f} A_{SN}^{l-f} (1 - A_{SN})^f$$

In the above equation, $\binom{l}{l-f} = \frac{l!}{(l-f)! \cdot f!}$, which is the combination function. The availability is a sum from zero to $n-m$ failures of the combinations of $l-f$ storage nodes that are available and f storage nodes that are unavailable. To use this formula, a single availability value must be used for the set of storage nodes. This value, A_{SN}, is chosen to be the minimum availability of the entire set of l nodes that hold data fragments. Choosing the minimum value ensures that the resulting calculation will be a conservative estimate, since there are typically less severe consequences to errors of this type.

In the example of Table 3.2, d1 would have an availability of 0.99, and d2 would have an availability of 0.99. These numbers would then be combined to form the availability of the overall dataset as: $0.99 \cdot 0.99 = 0.9801$. Again, this method of combining the data distributions' availability numbers is conservative. To illustrate these two sources of conservatism, consider the true availability calculation for d2:

$$\begin{aligned} AV_{d2} &= 0.1 \cdot 0.95 + 0.9 \cdot 0.05 + 0.9 \cdot 0.95 \\ &= 0.995 \end{aligned}$$

First, we see that the actual availability is 0.995, which is greater than the estimate of 0.99, a result of using the lower storage node availability number. Second, the dataset availability (using the more accurate estimate) is: $0.99 \cdot 0.995 = 0.9851$. This dataset availability is still less than that produced with the detailed model because it neglects the overlap of storage node sn2.

3.1.3 Improvements and related models

There are a number of potential improvements that could be made to the above models. For example, in both models, independence of storage node failures is assumed, but current research by Schroeder and Gibson [2007] and Pinheiro et al. [2007] suggest this is not an accurate assumption. Nicola and Goyal [1990] examined correlated failures across multiversion software, using a Beta-Binomial distribution as a way to model correlated failures. Another approach would be to attempt to model the sources of correlated failures such as dependencies on a common network, power, or

physical topology. Asami [2000] looked at a connectivity model with "and/or" nodes as a way of estimating the availability of data in a large storage system. Obtaining accurate measures of device availability and the level of correlation can be difficult. Brown and Patterson [2000] suggest creating availability "benchmarks" via fault injection to quantify system availability, and Douceur and Wattenhofer [2001a,b] measure and model the availability of 50,000 desktop computers as part of the Farsite project. Both of these could provide a starting point for creating more accurate models. AFRAID [Savage and Wilkes, 1996] was a system that sought to trade off availability for performance by delaying RAID-5 parity updates until idle periods.

Another area of improvement would be extending the availability modeling to include not just fractional availability but also a measure of frequency and duration of outage. This is particularly useful for examining penalties for outages because there is both a fixed and a variable component to the penalties that are incurred. For example, an outage is likely to cause applications to fail, and there is a non-zero cost (time) to restart them, so while one outage per year of 8.8 hours has the same fractional availability as a ten minute outage once per week, the penalties can be considerably different.

3.2 Capacity

Tracking capacity utilization in a storage system is important because it is a limited resource. Storage nodes have a limited amount of raw capacity that can be used to store data fragments. The capacity model calculates the capacity used on each storage node (in absolute terms and also as a fractional utilization), the storage "blowup" of each dataset, and the total storage system capacity used (absolute and fractional). The main use of these metrics by the administrator is to ensure some minimal amount of space remains free to accept new data (system-wide utilization) and potentially to charge end users for the capacity they consume (dataset size and blowup). The system also uses the capacity information to ensure a given configuration is feasible (i.e., the capacity consumed on each storage node is less than or equal to its capacity).

Calculating the capacity metrics is relatively straightforward, relying on the (raw) size of a dataset and the data encoding used to store it. The dataset *blowup* is the ratio of the stored data size to its raw size. It can have values of one or greater, assuming no compression is used. The size increase comes from the capacity overhead of storing redundant information as a part of the data encoding scheme. For example, a 1-of-3 scheme replicates data three times, giving a blowup

of three. The formula for blowup is:

$$Blowup = \frac{n}{m}$$

where m and n are the data encoding parameters.

To calculate the capacity utilization of storage nodes, begin by noting that the total capacity consumed by a dataset is:

$$\begin{aligned} Consumption &= Size \cdot Blowup \\ &= Size \cdot \frac{n}{m} \end{aligned}$$

This consumed capacity is divided equally among all storage nodes that hold fragments from this dataset (l). Therefore, the capacity consumed on a given storage node by a particular dataset (assuming the dataset has fragments on this node) is:

$$NodeConsumption = Size \cdot \frac{n}{m \cdot l}$$

This consumption is summed across all datasets to determine the total usage of each node. In the case of datasets that have more than one data distribution (due to an in-progress data migration), the consumption of each distribution is calculated separately and added to the total because all distributions must be able to store the entire dataset.

The system capacity consumption is just a sum of the usage on the individual storage nodes, and the fractional utilization is:

$$SystemCapUtilization = \frac{\sum_i^{snodes} NodeConsumption_i}{\sum_i^{snodes} NodeCapacity_i}$$

3.3 Cost and power

While cost and power metrics have very little in common, their calculation is very similar, and they are implemented within the same model. If a storage node holds fragments from a dataset, it is included in both the cost and power totals for the system. The cost and power metrics are system-wide metrics, representing the total purchase cost or power consumption of the storage system (nodes). Storage nodes have attributes, provided in their description, corresponding to their cost (in dollars) and power consumption (in Watts). The model for power consumption is that the device is

either on or off, with no intermediate power states, and a storage node consumes the same amount of power regardless of its workload. Since a node that stores data is assumed to be powered, the total system power metric is created by summing the power attribute from each storage node that holds a data fragment from one or more datasets. The total system cost is generated in a similar manner based on the storage node's cost attribute.

The cost model only sums the device cost for storage nodes that hold data. This is an artifact of how the system models storage provisioning. Instead of dynamically adding and removing devices, the system begins at a maximum size, and if any storage nodes are not used in the final configuration, they are not considered a part of the final design.

3.3.1 Improvements and related models

While the cost of a storage node is an all-or-nothing metric, real-world power consumption is not. Devices consume varying amounts of power based on their load and potentially their configuration. Adding this to the power model would provide the potential to automatically influence or configure the device's power consumption to maximize utility. As an example, systems such as Hibernator [Zhu et al., 2005] or PARAID [Weddle et al., 2007] try to configure storage systems to minimize power consumption while still meeting performance goals.

3.4 Management complexity

The complexity of a storage system's design influences the ease with which it can be managed. For example, an administrator can easily reason about a simple design with few components and determine the impact of an outage or system change. As a design gets more complex, it may become more efficient, but it also becomes more difficult to manage because the administrator may have difficulty remembering how individual devices are configured or what data they store. This is an important consideration since, for the foreseeable future, some tasks will continue to be performed manually by the system administrator. A good metric for management complexity is difficult to define, but the impact of complexity on the administrator's time should not be neglected. As a proxy for overall complexity, a metric based on system configuration symmetry was created.

The system asymmetry metric is an integer, greater than or equal to one, describing the total number of unique storage node configurations that exist in the system. This model defines a storage configuration as the union of both the storage node description and the data distributions that it

Table 3.3: Asymmetry model example – This table shows a storage system with six storage nodes (of two different types) and three datasets. Each row denotes a single storage node and indicates the datasets whose fragments reside on that storage node. The final column indicates the "symmetry class" for that configured storage node. For two storage nodes to be considered the same, they must be of the same type and store fragments from the same datasets. In this example, the storage system has an asymmetry value of four.

Node type	ds1	ds2	ds3	Symmetry
A	×			a
A	×			a
A	×	×		b
B	×	×		c
B			×	d
B	×	×		c
			Asymmetry:	4

stores. For two configured storage nodes to be considered the "same" for calculating the asymmetry metric, they must be of the same type and store fragments from the same datasets. Table 3.3 shows an example storage configuration with six storage nodes and an asymmetry value of four.

3.4.1 Improvements and related models

Measuring management complexity as the number of distinct storage node configurations is rather naive. Its presence in this work is more to raise the issue of complexity than to provide a definitive solution for balancing it with other system metrics. The inspiration for some notion of configuration or topology symmetry came from Appia [Ward et al., 2002], an automated SAN designer. The authors found that they could quickly create SAN designs that were considerably less expensive than those created by hand. A major difference between the two, however, is that the manual designs tend to be symmetric. While this obviously helps the human designer in creating the initial design, it is also likely to benefit the system administrator tasked with maintaining the system. Finding the proper balance between system configurations that are easier to manage versus cheaper to purchase is an area that needs further research.

Measuring "management complexity" is difficult. There are many criteria that could affect the manageability of a system, and the model presented here utilized only one — distinct storage

node configurations. Even with this simple model, it is difficult to assess how this metric should be incorporated into configuration trade-offs. Two configurations are easier to manage than seven, but, to be useful, this metric needs to be compared and weighed against system performance, cost, and data protection. Due to the lack of an easily quantifiable way to make these trade-offs, the asymmetry metric is not used in the utility experiments. Even though it may be difficult to use in making trade-offs, it may still prove useful as a constraint, to limit the storage configuration's complexity.

3.5 Performance

A storage system's performance is of significant concern to both system administrators and end users. The performance of a workload is described by a family of metrics, and each metric in that family is important to particular classes of workloads. The metrics analyzed by this performance model related to workloads include:

Request latency: The latency of a request is the time from when the request in generated at the client until the response(s) are received from storage nodes. Certain classes of workloads, such as online transaction processing (OLTP), tend to be latency sensitive. These applications typically have end users working interactively with the application, and they are forced to wait until storage requests complete before they can proceed.

Bandwidth: The bandwidth is the rate at which data is stored and retrieved from the storage system, measured as the amount of data transferred over some time frame (e.g., in units of MB/s). Bandwidth is important for data processing applications that must stream through a large volume of data.

Throughput: Throughput is similar to bandwidth, but it is measured as a number of I/O operations per unit time (e.g., IO/s).

The model analyzes the flow of requests generated by each workload, determining the load placed on system components, including the client CPU, client network interface, storage node network interface, and the storage node's disk. This analysis accounts for the chosen data distribution, calculating the (extra) load generated from the data encoding and directed at particular storage nodes.

3.5.1 Queueing model

The model chosen to estimate system performance is a closed-loop queueing model, and all queues are first-come-first-serve (FCFS). The benefit of using a closed-loop model is that it models the back-pressure that a slow storage system would cause on workloads. This back-pressure allows the modeling of interactions (interference) between workloads. Intuitively, most applications and operating systems have a limit to the number of simultaneous requests that they may have outstanding to a storage system. The longer a storage system takes to respond to requests, the fewer that will be completed in a fixed period of time.

A queueing model is defined by its request stream(s), which are generated by the workloads, and the queueing centers, which are the various system components that process I/O requests as they move through the system. The workload definition contains the following attributes that are used by the queueing model:

Multiprogramming level: The multiprogramming level of the workload is the maximum number of data requests that the workload can have outstanding simultaneously. Higher values increase the parallelism for the workload.

Think time: The think time is the average time a given request thread (terminal user in queueing theory) waits between the completion of a request and issuing the next. This would correspond to an application's processing time between I/O requests.

I/O size: This is the size (in bytes) of an I/O request from the application. The model uses the I/O size in determining the demand placed on system resources. For example, larger requests consume more network and disk time.

Read fraction: This is a value, between zero and one, inclusive, that determines the fraction of a workload's I/O requests that are read operations. The remainder are writes. Due to the data encoding schemes, read and write operations place different demands on the network and storage nodes because they transfer different amounts of data and interact with different numbers of storage nodes.

Random fraction: This is the fraction of requests that are not contiguous with the previous I/O request issued by a workload. This value, between zero and one, inclusive, influences the disk service time, because a non-contiguous request would likely incur additional positioning time over one that is part of a sequential stream.

Both clients and storage nodes contain a network interface that serves as a queueing center. The network interfaces are defined by the following attributes:

Bandwidth: This is the maximum bandwidth that the interface can transmit. For example, a 1 Gb/s Ethernet network would be modeled with a bandwidth of 125 MB/s.

Latency: Each network request has some overhead processing delay that is modeled as a fixed additional service time for each request.

Storage nodes are modeled with a single disk resource that has the following attributes:

Bandwidth: This is the sequential streaming bandwidth from the disk. Large, sequential I/O requests would approach this transfer rate.

Latency: This is the average positioning time for non-sequential accesses (i.e., the latency to the first byte transferred).

The above attributes are defined for each workload, client, and storage node independently, allowing configurations with heterogeneous storage components to be analyzed.

The design of the queueing model used here is based on the work of Thereska et al. [2006], with a few modifications to better fit the goals of this project. Thereska et al. used bounds analysis to locate system performance bottlenecks. However, bounds analysis is insufficient for capturing the interactions between the different workloads. Instead, the queueing model is solved using the Mean Value Algorithm [Gunther, 2005, p138] with the PDQ [Gunther and Harding, 2007] model solver. The PDQ solver calculates the relevant performance metrics based on the demand that each workload places on each queueing center (i.e., the components that service requests such as the network interfaces, CPUs, and disks).

Calculating service demand

The key to building the queueing model is determining the demand placed on each queueing center by the workloads. Demand is typically measured as the amount of time required to service each request. This demand can be broken into the product of the number of times a request visits the queueing center and the service time during each visit: $D = V \cdot S$. Breaking the demand up into these two components allows the demand to be calculated from the flow of requests (to calculate the visits) and the service time when a request is received.

Visits per request The easiest way to determine the number of visits per request is by following a typical request through the storage system. The number of visits is normalized to the number of workload requests. For the following examples, the dataset is stored in a single data distribution as m-of-n declustered across l, meaning that each data block is divided into n fragments when it is written, and the n fragments are spread (on a per-block basis) across a potentially larger set of storage nodes, l. Any m of the n blocks can be used to reconstruct the data block on a read. Due to differences between requests, reads and writes will be examined separately.

For a read, issued by a particular workload, the request is first sent by the client associated with that workload to m of the storage nodes in the dataset's data distribution. Assuming an even distribution of the fragments across the storage nodes in the data distribution, the probability that a given one of the l nodes is accessed by the read request is:

$$\begin{aligned} P_{read_snode} &= V_{read_snode} \\ &= \frac{m}{l} \end{aligned}$$

At this point, only if the request misses in the storage node's cache will it incur a disk access. This work has not modeled the storage node cache, instead using a fixed 80% hit rate for all requests. This results in 20% of requests sent to the storage node requiring a disk access. Therefore, the number of visits to the disk is:

$$\begin{aligned} V_{read_disk} &= V_{read_snode} \cdot (1 - HitRate) \\ &= V_{read_snode} \cdot 0.2 \\ &= \frac{0.2 \cdot m}{l} \end{aligned}$$

All storage node requests, whether they hit in cache or require a disk access, send data across the network to the client. Non-data transfers (e.g., the initial read request) are neglected by the model.

$$\begin{aligned} V_{read_Snet} &= V_{read_snode} \\ &= \frac{m}{l} \end{aligned}$$

On the client side of the storage system, read requests from all m fragment requests must be received on the client's network interface, but this is modeled as a single request with the size being

influenced by m:

$$V_{read_Cnet} = 1$$

and a single data block decode of the fragments is performed:

$$V_{read_cpu} = 1$$

A similar analysis produces the number of visits to each resource for writes. A data block is first encoded into n fragments,

$$V_{write_cpu} = 1$$

and all n fragments are transmitted via the client's network interface in a single operation:

$$V_{write_Cnet} = 1$$

There are l storage nodes that are a part of the data distribution, and n of them receive fragments from a given block:

$$V_{write_snode} = \frac{n}{l}$$
$$V_{write_Snet} = \frac{n}{l}$$

Again, 80% of the requests are assumed to be absorbed into the cache and later overwritten or destaged to disk during idle periods, while the remaining 20% are handled immediately. The background destaging is not directly modeled.

$$\begin{aligned} V_{write_disk} &= V_{write_snode} \cdot (1 - HitRate) \\ &= V_{write_snode} \cdot 0.2 \\ &= \frac{0.2 \cdot n}{l} \end{aligned}$$

The number of visits to each queueing center has been calculated separately for read and write operations. To get the overall number of visits to a queueing center, V_{read} and V_{write} need to be

combined based on the read/write ratio of the workload:

$$V = ReadFrac \cdot V_{read} + (1 - ReadFrac) \cdot V_{write}$$

This calculation is performed for each queueing center.

Service time per visit Calculating the service time at each queueing center can be considerably more tedious than calculating the visits. The visits were determined by the request flow from only the workload being analyzed, but the service time can be affected by the other workloads as well.

A request uses some amount of CPU time for encoding or decoding the data block. This time is directly specified in the client description, and the model does not distinguish between the type (read or write) or size of the request. It also does not model the effects of different encoding schemes on CPU demand.

The client network service time is calculated as a linear model based on the size of the request:

$$S_{Cnet} = Latency + \frac{Size}{Bandwidth}$$

The latency and bandwidth are specified in the client description. The size of the network request is influenced by the workload I/O size and the type of the I/O. Reads transfer exactly *IOSize* bytes while writes transfer $IOSize \cdot \frac{n}{m}$ due to the data encoding.

$$\begin{aligned}
S_{Cnet} &= Latency + \frac{Size}{Bandwidth} \\
&= ReadFrac\left(Latency + \frac{ReadSize}{Bandwidth}\right) + (1 - ReadFrac)\left(Latency + \frac{WriteSize}{Bandwidth}\right) \\
&= Latency + \frac{ReadFrac \cdot ReadSize + (1 - ReadFrac) \cdot WriteSize}{Bandwidth} \\
&= Latency + \frac{ReadFrac \cdot IOSize + (1 - ReadFrac) \cdot \frac{IOSize \cdot n}{m}}{Bandwidth}
\end{aligned}$$

The usage of the storage nodes' network interface is calculated using the same linear model, but the amount of data transfered is always a single encoding fragment:

$$\begin{aligned} S_{Snet} &= Latency + \frac{FragSize}{Bandwidth} \\ &= Latency + \frac{IOSize}{Bandwidth \cdot m} \end{aligned}$$

The service time for a storage node's disk is based on an initial positioning time plus transfer time. If the current request is contiguous with the previous request, the initial positioning time is omitted. For a storage node that has only one workload (and data distribution) accessing it, the initial positioning time would be used based on the workload's $FracRand$ that is specified in the workload description, leading to a overall service time of:

$$S_{disk} = RandFrac \cdot Latency + \frac{FragSize}{Bandwidth}$$

In general, there is more than one workload accessing a storage node, complicating the calculation regarding the fraction of requests that are sequential. For a request to be considered sequential in this model, it must have come from the same workload (and data distribution) as the previous, and only then, $(1 - RandFrac)$ of the I/Os are sequential. The fraction of requests that are sequential are: $\frac{1-RandFrac}{Workloads}$, assuming that the request rates are approximately equal across all workloads that access the storage node. For unequal request rates, the sequentiality of the slower workload will likely be overestimated while the value for the faster is underestimated. Unfortunately, to know the actual request rates (and their ratio), the queueing model must be solved. This unfortunate dependency is the reason for this approximation. This changes the Service time to be:

$$\begin{aligned} S_{disk} &= RandFrac' \cdot Latency + \frac{FragSize}{Bandwidth} \\ &= \left(1 - SeqFrac'\right) \cdot Latency + \frac{FragSize}{Bandwidth} \\ &\approx \left(1 - \frac{1 - RandFrac}{Workloads}\right) \cdot Latency + \frac{FragSize}{Bandwidth} \\ &\approx \left(1 - \frac{1 - RandFrac}{Workloads}\right) \cdot Latency + \frac{IOSize}{Bandwidth \cdot m} \end{aligned}$$

In the above formulae, *Workloads* is the number of unique data distributions that access the storage node. This may be greater than the true number of workloads, if a data migration/re-encoding is

Table 3.4: Summary of queueing model – This table lists the formulae for calculating the average number of visits and service time (the product of which is the demand) placed on each queueing center.

Queueing center	Visit ratio	Service time
Client CPU	1	$CPUTime$
Client network	1	$Latency + \frac{ReadFrac \cdot IOSize + (1-ReadFrac) \cdot \frac{IOSize \cdot n}{m}}{Bandwidth}$
Storage network	$\frac{ReadFrac \cdot m + (1-ReadFrac) \cdot n}{l}$	$Latency + \frac{IOSize}{Bandwidth \cdot m}$
Storage disk	$0.2 \cdot \frac{ReadFrac \cdot m + (1-ReadFrac) \cdot n}{l}$	$\left(1 - \frac{1-RandFrac}{Workloads}\right) \cdot Latency + \frac{IOSize}{Bandwidth \cdot m}$

being modeled.

Table 3.4 summarizes the formulae used to calculate the visits and service time at each resource. The product of these two is the service demand that can be used to solve the model. The demand must be calculated for each workload/data distribution and resource separately.

Solving via MVA

The performance model uses the Mean Value Algorithm (MVA) [Gunther, 2005, p138] to solve the queueing system. MVA is an iterative algorithm that provides a method to estimate the steady-state characteristics of a closed queueing system. The insight behind MVA is that a request entering a queue can see only a bounded number of jobs ahead of it. Because the job, itself, is just entering the queue, for a workload with a multiprogramming level of N, there can be at most $N-1$ jobs ahead of it. In steady-state, the number of jobs already waiting is the same as the average number of jobs at the queue in a system that has multiprogramming level of $N-1$. This leads to the following iterative algorithm that is repeated for each n from 1 to N. The response time, R_k, for each of the K queueing centers is calculated from their demand, D_k, and queue length, Q_k, with $n-1$ jobs:

$$R_{n,k} = D_k + D_k \cdot Q_{n-1,k}$$

These response times are summed to determine the overall system response time:

$$R_n = \sum_{k=1}^{K} R_{n,k}$$

This system response time is used to calculate the system throughput, including any think time, Z:

$$X_n = \frac{n}{R_n + Z}$$

The throughput and the individual queue response time can be used to estimate the new queue length:

$$Q_{n,k} = X_n \cdot R_{n,k}$$

There is also an approximate MVA algorithm that is based on the observation that the queue length, $Q_{N,k}$, is proportional to N when N is large. This allows an iterative, approximate solution that avoids the loop over N, instead looping until an accuracy tolerance has been reached.

Both the exact and approximate MVA algorithms are implemented in the PDQ [Gunther and Harding, 2007] analysis software that is used by this performance model. The exact algorithm is used when there are three or fewer workloads, and the approximate algorithm is used otherwise.

3.5.2 Improvements and related models

The model used to estimate system performance takes a number of shortcuts for ease of implementation. The model is primarily concerned with modeling the main data flow between clients and storage nodes. As such, it does not model interactions with the metadata server, the initial request/final acknowledgement of data operations, or overheads associated with RPC transactions. The network modeling also does not account for full duplex links, but this could be added by representing each network interface as a pair of queues, one for transmits and one for receives. The storage node cache model is also rather simplistic, using a fixed hit ratio. The CPU overhead of encode and decode operations is also modeled as a fixed service time. In reality, the size, request type, and encoding influence this cost.

The queueing model has no notion of a request waiting in multiple queues simultaneously. In the actual system, requests wait and are serviced in parallel at the storage nodes, but the queueing system models this as a series of sequential operations, increasing the predicted request latency.

Many researchers have developed performance models for storage systems. A number of techniques have been employed, including ad-hoc analytical models [Geist and Trivedi, 1993], queueing models [Uysal et al., 2001; Varki et al., 2004], empirical models [Anderson, 2001], and relative models [Mesnier et al., 2007], to name a few.

Meyer [1980] stresses the importance of modeling not just fault-free performance but also the performance for systems in degraded states. This performability modeling is increasingly important as systems increase in size and move to providing high levels of service using large numbers of lower performing (and less reliable) components. Extending the performance model to generate a range of performance metrics that account for the degraded data distributions can lead to better predictions of overall system utility. Currently, the performance and any utility calculation that is based on it assumes fault-free performance even though using m-of-n encodings provide a large number of possible degraded but operational states, many of which would have lower utility than predicted by a fault-free analysis. Depending on the particular utility function, it may be the case that relatively few of the degraded states need to be analyzed to obtain an accurate estimate. Techniques such as those presented by Alvarez et al. [2001a] could be useful for this analysis.

3.6 Reliability

Reliability, $\mathrm{R}(T)$ is the probability that a system, functioning at $t = 0$, will still be operating properly at $t = T$. Typically, instead of modeling the actual reliability function, it is reduced to a single metric, such as the mean time to failure (MTTF) or the annual failure rate (AFR). While the AFR and MTTF are reciprocals of one another, this document will prefer using the AFR numbers because they are slightly more intuitive for judging risk.

This system model uses AFR attributes from storage nodes and a dataset's data distribution(s) to calculate the AFR of the dataset. This AFR can be used as a measure of risk of data loss, likely resulting in financial penalties related to recreating or restoring data from an external backup.

3.6.1 Markov model

The reliability metric for a dataset is calculated using a Markov chain with states representing the number of storage node failures. The chain terminates in an absorbing state that indicates the loss of data for the dataset being modeled. The chain structure for a given dataset is determined by the data encoding. Since an m-of-n encoding can tolerate $n - m$ failures and continue functioning, the

chain has $n-m+2$ total states to account for failure-free operation as well as an absorbing state that accounts for the final, catastrophic failure. For simplicity, the states will be referred to by the number of failures they represent (i.e., $f = 0 \ldots (n-m+1)$).

State transitions within the model are due to storage node failures and data re-encoding operations that return the encoding to its failure-free state. Failure transitions move the model from the current state, f, to the state $f+1$, indicating one additional failure. Repair operations are modeled as a complete re-encoding of the dataset. As such, multiple failures can be repaired simultaneously. Using this model, the repair operation from each state, $0 < f \leq n-m$, transitions back to the fault-free state, $f = 0$.

The failure rates are modeled as a base failure rate, λ, for a single storage node, multiplied by the number of functioning storage nodes in the data distribution. For example, a dataset with an encoding of m-of-n declustered across l, there are a total of l storage nodes that hold data from this dataset in the failure-free state. With one failure, there are only $l-1$ nodes remaining that can fail and affect this dataset. The base failure rate, λ, is the maximum failure rate of any of the l storage nodes. This leads to a conservative estimate of the dataset's reliability in cases where heterogeneous storage nodes are used.

The repair rate, μ, used in the model is influenced by the amount of time required to re-encode the dataset. This time is estimated based on the amount of data that must be re-encoded and a fixed fraction (5% in all experiments) of the streaming bandwidth of the storage nodes. The actual repair rate is:

$$
\begin{aligned}
DataToMove &= \frac{DatasetSize \cdot n}{m \cdot l} \\
\mu &= \frac{DataToMove}{EffBandwidth} \\
\mu &= \frac{DatasetSize \cdot n}{m \cdot l \cdot BW \cdot 0.05}
\end{aligned}
$$

The bandwidth (BW) used above is the minimum streaming bandwidth of the l storage nodes, again producing a conservative estimate. An example Markov chain for an $(n-2)$-of-n declustered across l (e.g., 3-of-5) encoding is shown in Figure 3.1.

Once the chain has been constructed, it must be solved to determine the time until the absorbing state is entered (i.e., the time until data loss). Using the transformation discussed by Pâris et al. [2006], the chain can be altered to remove the absorbing state, allowing traditional solution techniques to be used. The insight is that instead of viewing the process as a one-time occurrence, it

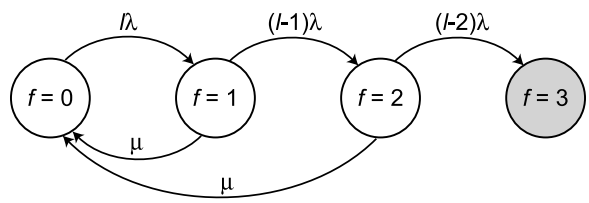


Figure 3.1: Example Markov chain for reliability model – This figure shows an example Markov chain used to calculate the reliability for an $(n-2)$-of-n declustered across l data encoding. The $f = 3$ state is an absorbing state, indicating data loss.

can be modeled as a process that immediately restarts in the initial state after a data loss. With this view, the chain is modified such that the transition into the absorbing state is redirected to the $f = 0$ state, and the quantity of interest is the rate at which this transition is followed. For the example in Figure 3.1, this amounts to redirecting the arc between $f = 2$ and $f = 3$ to point from $f = 2$ to $f = 0$. This removes state $f = 3$. The rate that the $f = 2$ to $f = 0$ transition is followed is $(l-2)\lambda$ times the probability of being in state $f = 2$. The goal, then, is to solve for the limiting probabilities (the P_i's) of being in each state. This can be done using the "balance equations," which are based on the concept that every transition into a state must be balanced with a transition out (e.g., the transition rate into $f = 0$ equals the transition rate out of $f = 0$). For a system with three states (as in the example above), this produces two independent equations. Combining this with the knowledge that the sum of the limiting probabilities is one provides three independent equations and three unknowns (the limiting probabilities). The system of equations for Figure 3.1 is shown below.

$$
\begin{aligned}
l\lambda P_0 &= (l-1)\lambda P_1 + \mu P_1 \\
(l-1)\lambda P_1 &= (l-2)\lambda P_2 + \mu P_2 \\
\sum_{i=0}^{2} P_i &= 1
\end{aligned}
$$

This system of linear equations is solved via LR decomposition[1] to determine the limiting probabilities (the P_i's). The failure rate for the example above is then:

$$FailureRate = P_2 \cdot (l-2)\lambda$$

[1]This is also knows as LU decomposition and is an application of Gaussian Elimination.

3.6.2 Improvements and related models

The reliability model makes a number of simplifications that limit the experimentation for which it can be used. The model assumes a single failure rate for each storage node. This simplifies the model considerably, allowing the states to represent the number of failures instead of having one state for each possible combination of failures. There may be some benefit to using combinations of reliable and unreliable storage nodes together, potentially saving money on less expensive nodes while having the reliability boosted by a few with high reliability. Unfortunately, this cannot be evaluated with the current, single failure rate model.

Currently, the repair rate is based on using a fixed fraction of the storage node bandwidth to carry out repair operations. Based on the system model, however, a failure may not be repaired, or it may be repaired and migrated to an entirely different data distribution. It is unclear how this should be represented within the model. One approach would be to not model repair at all. This would produce reliability values that assumed a data distribution would accumulate failures until it finally fails. Such a model would be conservative because it is highly likely that a future tuning operation in the storage system would make repairs before data is actually lost.

Reliability models of storage using Markov models are not new. Amiri and Wilkes [1996] produced similar models for availability and reliability. Gibson [1991] examined the reliability of RAID arrays. Also, Burkhard and Menon [1993] used a Markov model to examine the reliability of RAID groups as the size of the group and number of parity disks is adjusted. Lee and Leung [2002] examined the reliability of a video-on-demand service that used a cluster of servers that stored data with m-of-n schemes.

There are a number of other failure modeling options that could be used in this architecture. For instance, Apthorpe [2001] examined fault trees and event trees for modeling reliability.

3.7 Other models

This chapter has provided a description of the current set of models and metrics that are available for use in experiments. The tool's architecture allows additional models to be easily added, expanding the vocabulary of objectives that can be expressed using utility functions. This section discusses potentially interesting additional models that could be added.

3.7.1 Physical space

Data center floor space is a scarce resource. The total square footage for computer equipment cannot be easily expanded, and many organizations have very limited space available. A metric that describes the amount of data center space occupied by a storage solution would be useful for automatically trading off potential storage form factors (e.g., rack-mounted vs. free standing) or ensuring a storage system fits within a specified equipment space (e.g., twelve rack units or 12 U). This metric could be used to assign a cost per rack unit that the system consumes, allowing a utility function to account for the "rent" of the physical space over the system's expected lifetime.

3.7.2 Robustness

In general, it is quite difficult to model storage systems accurately. Approximations are made for the sake of modeling ease, some components are poorly understood or their construction is unknown, and workloads may change over time. All of these issues lead to uncertainty in the predictions of system metrics. A measure of the system configuration stability could be useful.

Consider the following two scenarios. The first is optimal according to the provided metrics and objectives, but with a small change to any workload, the overall system poorly satisfies its objectives. In the other scenario, the configuration may not quite be optimal, but the configuration is resilient against small workload changes. This second configuration would be much more robust and potentially more desirable.

One way to quantify this as a system metric would be as a number that represents the amount by which a workload could change (e.g., its intensity, I/O size, or R/W ratio, etc.) before the observed performance would change by some fixed percentage (e.g., 10%). This would be a metric representing the performance stability of a configuration.

3.7.3 Similarity

There is a certain learning curve associated with new hardware and configurations. During this initial time period, a system administrator may be slower at responding to problems and less effective at managing the system. Because of this learning curve, it may be beneficial to rank new systems and configurations based on their similarity to a current or previous system with which the administrator is proficient.

Quantifying the notion of similarity is difficult because it should account for hardware commonalities such as network interfaces and form factors as well as more difficult to quantify properties such as system failure semantics and configuration techniques.

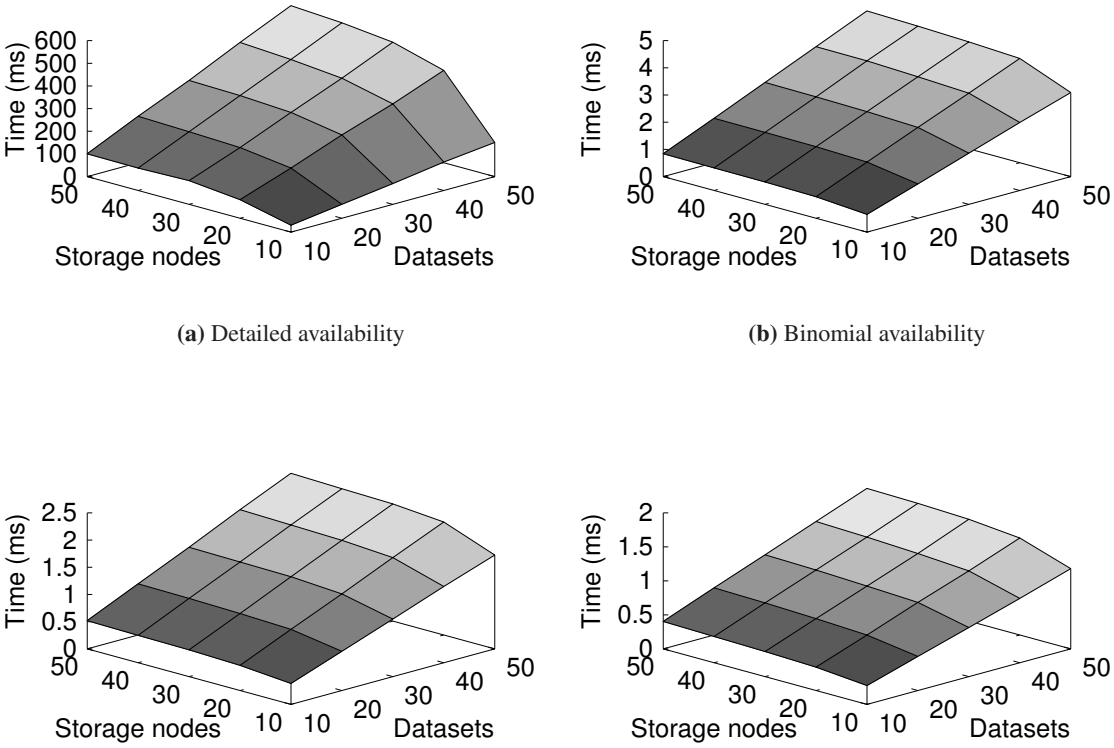
3.8 Summary of models

The models described above were chosen to produce a reasonable selection of metrics that allow a thorough investigation into using utility to control a distributed storage system. While the models have not been validated against a live system and make numerous simplifications that could impact their accuracy, the spectrum of modeling techniques and complexity provides a strong indication that conclusions based on these models will translate well to a system based on validated models.

3.8.1 Model performance

The time required for the models to analyze a storage configuration and calculate its metrics directly impacts the speed at which both provisioning and tuning operations can be planned. To investigate the time required to analyze storage configurations, a baseline configuration was created and scaled to various numbers of workloads and storage nodes. Both the number of workloads and storage nodes were scaled between 10 and 50, independently, in steps of 10, creating 25 total system sizes. Each of these 25 system sizes was analyzed by each of the system models to produce the measurements in Figure 3.2. The data was generated by repeatedly randomizing the storage configuration, allowing individual datasets to use up to eight storage nodes each ($1 \leq l \leq 8$). For each system size, configurations were generated and analyzed for 60 seconds, tracking the number of configurations that were evaluated and the amount of time spent in the models (as opposed to creating the random configurations). The figure plots the average of these trials, and the 95% confidence interval for all data points is within 10%.

Figure 3.3 provides a composite picture of the time required to calculate system metrics using the models presented here. Again, random configurations of the 25 different sizes were analyzed, averaged, and plotted. For all points, the 95% confidence interval is within 3% of the average. The models used for this graph are the binomial availability, capacity, power and cost, performance, reliability, and symmetry models. This is the same selection of models that are used in experiments throughout the rest of the document.



(a) Detailed availability

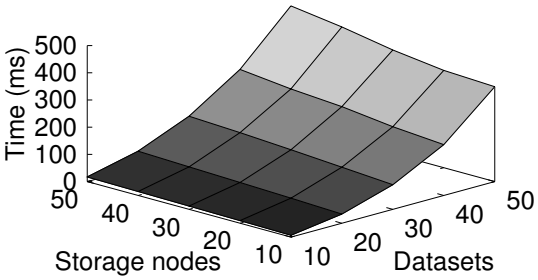
(b) Binomial availability

(c) Capacity

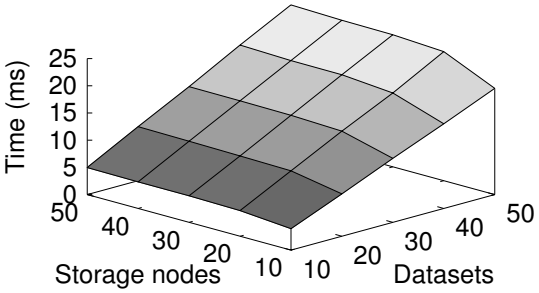
(d) Power and purchase cost

Figure 3.2: System model performance – These figures show the performance of the individual system models as a function of the size of the storage system being evaluated. The storage system sizes range between 10 and 50 storage nodes with 10 to 50 datasets in the system. There is always one workload and one client for each dataset.

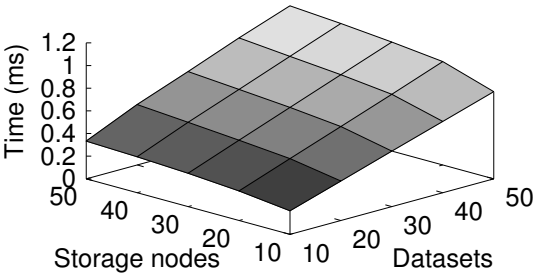
The largest configuration examined, 50 storage nodes and 50 datasets, required approximately 0.46 seconds to evaluate. The performance model is the limiting factor for analyzing configurations, consuming 96% of the computation time.



(e) Performance



(f) Reliability



(g) Design symmetry

Figure 3.2: System model performance (cont.) – These figures show the performance of the individual system models as a function of the size of the storage system being evaluated. The storage system sizes range between 10 and 50 storage nodes with 10 to 50 datasets in the system. There is always one workload and one client for each dataset.

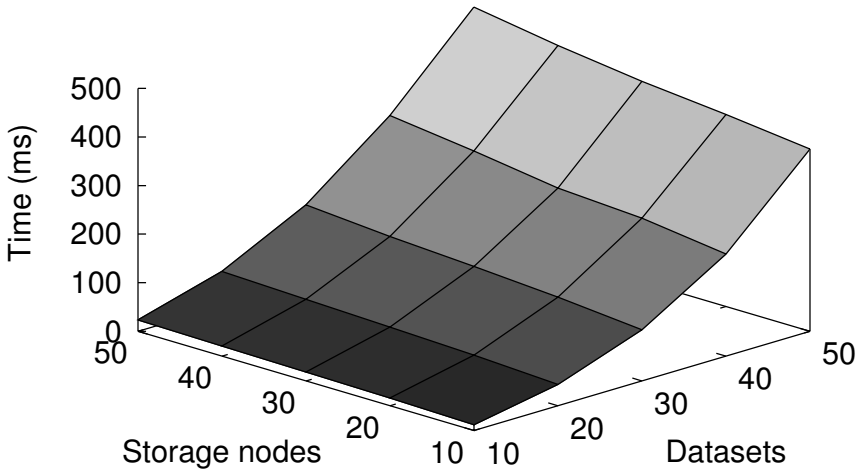


Figure 3.3: Total time to calculate system metrics – This figure shows the total amount of time required for the system models to analyze a storage configuration. The availability model used here is the binomial model. The storage system sizes range between 10 and 50 storage nodes with 10 to 50 datasets in the system, and there is always one workload and one client for each dataset. The performance model takes the majority of the time required by the entire set of models.

Chapter 4

Utility

Every decision has, and is predicated upon, a set of consequences. These consequences have their benefits and costs, and making the best choice amounts to choosing the one with the largest net benefit. However, analyzing alternatives frequently requires comparing and making trade-offs across dissimilar metrics. For storage systems, these metrics can include system (purchase) cost, power consumption, performance, and reliability. To automatically provision and tune a storage system, a tool must be able to navigate these choices in an automated manner, weighing the alternatives along each of these system attributes.

4.1 Overview of utility

Utility is a method for weighing the consequences of decisions. It provides a framework for collapsing dissimilar metrics into a single value so that alternatives can be evaluated. Keeney and Raiffa [1993] provide a background on utility and how it can be used to evaluate decisions. Their book provides a much more in-depth and formal presentation of the concepts described below.

Each outcome of a decision can be assigned some utility value based on its consequences. The best choice is the one with the highest expected utility value. Utility can be either *ordinal* or *cardinal*. Ordinal utility specifies only that all alternatives are totally ordered by the utility value. That is, for each possible outcome, Y, if Y_i is preferred to Y_j, the utility value, u_i, is greater than u_j, for all i and j, but the magnitude of the difference between the utility values does not imply anything about the magnitude of preference. As an example, consider three utility values, $u_0 = 5$, $u_1 = 10$, and $u_2 = 11$, corresponding to outcomes Y_0, Y_1, and Y_2. With ordinal utility, all that can

be inferred is that Y_2 is preferred to Y_1, which is, in turn, preferred to Y_0. The magnitude of the differences between the values is not relevant. For cardinal utility, however, the magnitude of the utility difference is important. Cardinal utility opens up the possibility of reasoning about uncertain outcomes by calculating the expected value of utility. For example, a decision with two choices may have the following property: Alternative #1 produces a result with $u_1 = 5$, and alternative #2 produces a result of $u_2 = 11$ with probability 0.5 and a result of $u_2 = 3$ otherwise. Comparing the expected values of utility, $\mathrm{E}(u_1) = 5$ and $\mathrm{E}(u_2) = 0.5 \cdot 11 + 0.5 \cdot 3 = 7$ tells us that alternative two is preferable. While this work does not directly consider such uncertainty, the dynamic tuning relies on the ability to trade off utility over time in a similar manner.

A *utility function* maps outcomes to utility values. For example, an outcome, Y_i, is evaluated by some utility function, $\mathrm{U}(\cdot)$, into a utility value: $u_i = \mathrm{U}(Y_i)$. Typically, decision consequences are characterized by multiple attributes (e.g., different storage system metrics), $Y_{i,0}, Y_{i,1}, \ldots, Y_{i,n-1}$, with the utility function being a multi-dimensional function of the attributes: $u_i = \mathrm{U}(Y_{i,0\ldots n-1})$. Such arbitrary functions are typically very difficult to specify because of their high dimensionality. When specifying preferences for the various attributes, it is desirable to be able to do so independently of each other. For example, instead of trying to create a single function that represents all possible values of workload latency crossed with dataset availability, it is desirable to express preferences for latency and preferences for availability individually. At its extreme, this allows the system administrator to specify n 1-dimensional functions instead of a single function that is n-dimensional.

While there are several types of independence, this work makes use of *additive independence* because it simplifies the specification of utility. With additive independence, the preferences across the individual attributes are expressed as individual utility functions (e.g., $u_{i,0} = \mathrm{U}_0(Y_{i,0})$ and $u_{i,1} = \mathrm{U}_1(Y_{i,1})$). These individual functions are then combined, via weights (e.g., a_0 and a_1), to produce the final utility value:

$$u_i = \sum_{j=0}^{n-1} a_j \mathrm{U}_j(Y_{i,j}) + c$$

In order to use this form, it must be the case that the preferences for $Y_{*,j}$ are not affected by the value of any $Y_{*,k}, k \neq j$. While this condition does not, in general, hold if the Y's are assumed to be storage metrics such as latency or availability, independent functions can often be constructed from higher-level abstractions, examples of which will be discussed in Section 4.3.

4.2 Utility in computer systems

Kephart and Walsh [2004] present autonomic computing in artificial intelligence terms, comparing approaches that use if-then (ECA) rules, goal-based systems, and utility-based systems. They state that a utility-based approach has the ability to guide systems in situations where goal-based rules may conflict, because "Utility Function Policies allow for unambiguous, rational decision making by specifying the appropriate tradeoff." This need to make trade-offs has caused a surge in interest in utility for autonomic, or self-$*$, systems.

Both Irwin et al. [2004] and AuYoung et al. [2006] use utility to schedule batch computation. The former uses utility to schedule jobs, accounting for a decreased benefit as a job takes longer to complete. The latter work examines scenarios where a collection of job results are necessary before the submitter can act on them. This leads to a desire to schedule either most or none of a user's jobs.

Chen et al. [2004] combine dynamic pricing of web workloads with admission control to maximize the profits of a web service provider. The system trades off revenue from additional requests against potentially lower revenue from already-accepted requests that may be degraded by the acceptance of additional work. IBM Research has also investigated using utility to control web-based workloads. Walsh et al. [2004] use utility functions to assign servers between two different service classes (e.g., gold vs. silver customers). Tesauro [2005] uses reinforcement learning to shift resources between batch and interactive applications to best satisfy utility functions derived from SLOs for each application type. Kephart and Das [2007] apply utility to control the response time in IBM's WebSphere Extended Deployment.

Utility has also been used for automatically determining fidelity settings in multimedia applications [Ghosh et al., 2003; Lee et al., 1999]. This work examined trading off different fidelity axes such as color depth, frame rate, and frame size.

In more speculative work, Candea and Fox [2002] discuss the use of utility for designing systems, but they take a more traditional decision theory approach as opposed to using utility as a framework to automate the process. Kelly [2003] argues for using utility to provision systems in a Utility Data Center, using Integer Programming and the CPLEX solver.

4.3 Cost-based utility

While utility holds promise for allowing tools to automatically navigate trade-offs for system provisioning and tuning, the task of creating suitable utility functions is not trivial. Finding a method for

system administrators to communicate their preferences (or even determine what those preferences are) is a difficult task. While it has its shortcomings, business costs provide a reasonable starting point for creating utility functions, because they allow the direct, monetary effects to be captured.

Utility based on business costs has been used to design storage systems for dependability [Keeton et al., 2004; Keeton and Wilkes, 2002]. It has also been used to examine recovery after disasters [Gaonkar et al., 2006; Keeton et al., 2006]. The disaster protection and recovery solutions were designed to minimize the total expected costs, accounting for both the cost of the protection method(s) and the penalties that would be incurred during an event. Bhagwan et al. [2005] proposed using time-dependent utility functions to change how data is treated over its lifetime, creating a system for automating ILM.

Storage objectives come from many sources. The service level for each dataset and workload affect the overall satisfaction with the storage system. Additionally, there are system-wide metrics that are important, such as the system's purchase cost and power consumption. Each of these different sources of utility need to be combined to form a single utility function. Identifying sources that are independent of each other allows the specification to be simplified, and these independent sources can be combined using the formula from Section 4.1:

$$u = \sum_{j=0}^{n-1} a_j \mathrm{U}_j(Y_j)$$

To use this form, it must be possible to identify the (independent) attributes, Y, how they translate into utility, $\mathrm{U}(\cdot)$, and how they can be properly scaled relative to each other, using a. To identify the attributes for the utility function, one can look toward the items that affect business costs and revenues. Some common sources are applications' progress (e.g., database throughput or a streaming application's bandwidth), maintenance activity (e.g., repair costs or lost productivity due to outage), or direct infrastructure costs (e.g., purchase cost or power consumption). With an understanding of these contributors, a utility function can be constructed to express these business costs. For example, a storage system consuming w Watts of electricity at \$0.12 per kWh would have a utility function of:

$$U_{\text{power}} = \left(\frac{-\$0.12}{\text{kW}\cdot\text{hr}}\right) w$$

Once the attributes and utility functions have been identified, they must be scaled relative to each other to ensure that proper trade-offs are made. One of the obvious benefits to using business costs is that it provides a way to scale the individual utility functions. When using business costs, utility

is a rate of money per unit of time. In the example above, U_{power} is in units of dollars per thousand hours. The value of the a_j's should be chosen so that all utility functions have the same units. For the majority of examples presented here, dollars per year will be used. While this discussion has explicitly discussed the scaling factor, a, it will typically be incorporated into the construction of the corresponding utility function such that all utility functions will have the same units and can be summed directly.

4.3.1 Examples

The above discussion of using business costs to create utility functions was somewhat abstract. To make the process more concrete, three hypothetical examples are discussed below. The first scenario is an imaginary e-commerce web site, and it will serve as the basis for much of the evaluation of storage provisioning using utility. The second scenario, trace processing, provides an example of a non-linear relationship between utility and one of the system metrics. The third scenario provides an example, using a web server and database, where utility is tied to the performance of two different workloads in a dependent manner.

E-commerce web site Online retailers generate the majority of their revenue via customers placing orders on a web site. These orders are passed to a warehouse where employees package merchandise and ship it out to the customers. Using this business model, it is possible to create a utility function that describes the value of the storage system to the retailer.

Revenue is generated via the transaction processing workload from customers placing orders on the web site. Based on the average revenue per order, the fraction of transactions that are for new orders, and the average number of I/O operations per transaction, the administrator may determine that, on average, the company receives 0.1¢ per I/O that completes. This expression of revenue based on the throughput (in IO/s) must also account for the lack of orders should the system be unavailable:

$$\begin{aligned}\mathrm{U}_{\text{perf}}\left(IOPS_{WL}, AV_{DS}\right) &= (\$0.001)\, IOPS_{WL}\, AV_{DS} \left(\frac{3600\text{ s}}{\text{hr}}\right)\left(\frac{24\text{ hr}}{\text{day}}\right)\left(\frac{365.25\text{ day}}{\text{yr}}\right) \\ &= \left(\frac{\$3.2\text{x}10^4\text{ s}}{\text{yr}}\right) IOPS_{WL}\, AV_{DS}\end{aligned}$$

While the system is unavailable, the warehouse is unable to process and fill orders, and employees must work to fix the outage. This cost is estimated to be \$10,000 per hour.

$$\begin{aligned}\mathrm{U}_{\mathrm{av}}\left(AV_{DS}\right) &= \left(\frac{-\$10,000}{\mathrm{hr}}\right)\left(1-AV_{DS}\right)\left(\frac{24\ \mathrm{hr}}{\mathrm{day}}\right)\left(\frac{365.25\ \mathrm{day}}{\mathrm{yr}}\right)\\ &= \left(\frac{-\$8.8\mathrm{x}10^{7}}{\mathrm{yr}}\right)\left(1-AV_{DS}\right)\end{aligned}$$

The cost of losing the dataset from the primary storage system is estimated to be \$100 M.

$$\mathrm{U}_{\mathrm{rel}}\left(AFR_{DS}\right) = \left(-\$100\ \mathrm{M}\right)AFR_{DS}$$

Accounting for the cost of electricity to power the system, assuming it consumes P Watts of electricity:

$$\begin{aligned}\mathrm{U}_{\mathrm{pwr}}\left(P\right) &= \left(\frac{-\$0.12}{\mathrm{kWh}}\right)P\left(\frac{\mathrm{kW}}{1000\ \mathrm{W}}\right)\left(\frac{24\ \mathrm{hr}}{\mathrm{day}}\right)\left(\frac{365.25\ \mathrm{day}}{\mathrm{yr}}\right)\\ &= \left(\frac{-\$1.05}{\mathrm{W{\cdot}yr}}\right)P\end{aligned}$$

The price of electricity used here (\$0.12 per kWh) is considerably higher than the national average for industrial customers, which is \$0.0675 per kWh as of July 2007 [Energy Information Administration, 2007]. The generous amount used here should cover the extra power required for cooling and support equipment as well.

For provisioning scenarios, it is also important to account for the purchase cost, C, of the system, which is assumed to be amortized across a projected three year lifetime:

$$\mathrm{U}_{\mathrm{purchase}}\left(C\right) = \left(\frac{-C}{3\ \mathrm{yr}}\right)$$

The above four (independent) utility functions share the same units (dollars per year) and can be summed to produce the final utility function for this e-commerce scenario:

$$\begin{aligned}\mathrm{U}\left(IOPS_{WL}, AV_{DS}, AFR_{DS}, P, C\right) =& \mathrm{U}_{\mathrm{perf}}\left(IOPS_{WL}, AV_{DS}\right) + \mathrm{U}_{\mathrm{av}}\left(AV_{DS}\right)\\ &+ \mathrm{U}_{\mathrm{rel}}\left(AFR_{DS}\right) + \mathrm{U}_{\mathrm{pwr}}\left(P\right) + \mathrm{U}_{\mathrm{purchase}}\left(C\right)\end{aligned}$$

Trace processing Trace analysis is a common activity within the Parallel Data Laboratory at Carnegie Mellon. Typically, students use software to scan through previously captured file system or I/O traces, looking for insights or validation of new ideas. By examining this activity and making some simplifications, it is possible to construct a utility function that would pertain to trace analysis by the PDL's graduate students.

In this example, utility is described by the costs associated with the trace processing activity. These costs are mainly associated with the cost of the graduate students and system administrator, both of which are assumed to cost the university \$35 per hr[1]. Based on these numbers and some assumptions about the behavior of the people involved, it is possible to construct a utility function that relates costs with storage system bandwidth and data availability.

In this scenario, traces are analyzed as a whole, and students wait for the run to complete. This assumption allows the cost of a student's time to be calculated based on the time required to process a trace. This assumption could be relaxed to allow the student to perform other work (with some efficiency) during the trace processing or to create a decaying value (increasing cost) as a function of the slowdown of the task in a similar manner to what was used by Irwin et al. [2004]. Processing the trace is assumed to be I/O-bound, permitting the calculation of the time required to process the trace based on the bandwidth from the storage system and the size of the trace. This could be relaxed with little difficulty, resulting in a maximum benefit (minimum cost) at, and above, the point where the I/O system is no longer the bottleneck. An outage of the trace data set affects one student and the system administrator, both of whom will be occupied repairing the outage. There are 250 trace analyses per year, which is approximately one run per regular work day.

Given the above assumptions and the knowledge that, on average, a trace set is 27 GB in size[2], the following utility function can be created based on the bandwidth of the trace processing workload, BW_{WL}, (in MB/s):

$$\begin{aligned}
\mathrm{U}_{\mathrm{perf}}\left(BW_{WL}\right) &= StudentTimeSpentWaiting \\
&= \left(TimePerRun\right)\left(FrequencyOfRun\right)\left(StudentTime\right) \\
&= \left(\frac{27\ \mathrm{GB}}{BW_{WL}}\right)\left(\frac{250\ \mathrm{runs}}{\mathrm{yr}}\right)\left(\frac{-\$35}{\mathrm{hr}}\right)\left(\frac{1024\ \mathrm{MB}}{\mathrm{GB}}\right)\left(\frac{\mathrm{hr}}{3600\ \mathrm{s}}\right) \\
&= \left(\frac{-\$6.72\mathrm{x}10^{4}\ \mathrm{MB}}{\mathrm{yr{\cdot}s}}\right)\left(\frac{1}{BW_{WL}}\right)
\end{aligned}$$

[1]This cost includes not only salary or stipend but also benefits, including tuition.
[2]The average trace size was calculated from the PDL's trace repository.

In a manner similar to Patterson [2002], the cost of downtime as a function of availability can be expressed as:

$$\begin{aligned}
\mathrm{U}_{\mathrm{av}}\left(AV_{DS}\right) &= \left(DowntimeCost\right)\left(1-AV_{DS}\right) \\
&= \left(StudentTime + AdministratorTime\right)\left(1-AV_{DS}\right) \\
&= \left(\left(\frac{-\$35}{\mathrm{hr}}\right)+\left(\frac{-\$35}{\mathrm{hr}}\right)\right)\left(1-AV_{DS}\right)\left(\frac{24\ \mathrm{hr}}{\mathrm{day}}\right)\left(\frac{365.25\ \mathrm{day}}{\mathrm{yr}}\right) \\
&= \left(\frac{-\$3.94\mathrm{x}10^{5}}{\mathrm{yr}}\right)\left(1-AV_{DS}\right)
\end{aligned}$$

If the storage system should fail and lose the trace data, it is estimated that it would require approximately 15 hr of the administrator's time to either restore or re-acquire them from the original source. The expected cost due to this lack of reliability is related to the dataset's annual failure rate[3]:

$$\begin{aligned}
\mathrm{U}_{\mathrm{rel}}\left(AFR_{DS}\right) &= \left(\frac{-\$35}{\mathrm{hr}}\right)\left(15\ \mathrm{hr}\right)AFR_{DS} \\
&= \left(-\$525\right)AFR_{DS}
\end{aligned}$$

The power and purchase costs follow those of the e-commerce example, above:

$$\begin{aligned}
\mathrm{U}_{\mathrm{pwr}}\left(P\right) &= \left(\frac{-\$1.05}{\mathrm{W}\cdot\mathrm{yr}}\right)P \\
\mathrm{U}_{\mathrm{purchase}}\left(C\right) &= \left(\frac{-C}{3\ \mathrm{yr}}\right)
\end{aligned}$$

Also, as above, the component utility functions share a common set of units so they can be directly summed:

$$\begin{aligned}
\mathrm{U}\left(BW_{WL}, AV_{DS}, AFR_{DS}, P, C\right) = &\mathrm{U}_{\mathrm{perf}}\left(BW_{WL}\right) + \mathrm{U}_{\mathrm{av}}\left(AV_{DS}\right) + \mathrm{U}_{\mathrm{rel}}\left(AFR_{DS}\right) \\
&+ \mathrm{U}_{\mathrm{pwr}}\left(P\right) + \mathrm{U}_{\mathrm{purchase}}\left(C\right)
\end{aligned}$$

[3] One could also add a term to this equation for the lack of student productivity, but I will assume they can shift to another task during the outage.

Workload dependencies The above examples involved only a single workload and dataset, but most real-world scenarios involve multiple workloads and datasets in a given storage system. In the case where the applications are independent (e.g., running the e-commerce site and trace processing together), the components for utility could be summed independently, assuming a common currency such as dollars. Another situation of interest is one in which the workloads are not independent. Consider a web site where each page served to clients contains a mix of static and dynamic content. One example of this could be a site that provides stock market price quotations. The bulk of each web page is static, but the actual price quote changes each time the page is presented. This could be modeled as two storage workloads running on the web server (the storage client), accessing two different datasets. One is accessing files that serve as page templates, and the other is accessing a database of quotations. One component of utility in this scenario would likely be the performance (pages per second) of serving web pages. For a page to be served, I/Os from both workloads must be completed, and they must complete in a specific ratio. This would lead to a utility function of the form:

$$\mathrm{U}_{\mathrm{perf}}\left(IOPS_{WL_1}, IOPS_{WL_2}\right) = \left(ValPerPage\right) \min\left(IPP_1 \cdot IOPS_{WL_1}, IPP_2 \cdot IOPS_{WL_2}\right)$$

where IPP_j is the number of I/Os required from workload j for each page served. This equation states that the utility is limited by the slower (scaled) of the two workloads.

4.4 Priority-based utility

The above section advocated using business costs to construct cardinal utility functions, but there may be situations where there are no good cost metrics available or, perhaps, the administrator is unfamiliar with utility and would instead prefer to express his objectives as a strict set of priorities. Strict priorities can be used to compare two configurations, but it is difficult to assess "how much" better one configuration is than another. With cost-based utility, having twice the net income is likely to be approximately twice as good, but is meeting twice as many objectives twice as good? Instead of attempting to make judgements such as this, the following scheme uses only ordinal utility to evaluate configurations.

In a typical priority-based scenario, the system administrator has a set of prioritized objectives for the storage system such as:

1. **Data protection:** Achieve four nines of availability

2. **Performance:** Achieve 90 IO/s

3. **Capacity:** Minimize capacity consumption

The system should first spend resources to obtain high availability for the dataset. Next, obtain at least 90 IO/s, while not allowing the availability to slip below four nines. After achieving both of the first two objectives, work to minimize the capacity consumption (by choosing more space efficient data encodings).

Examining each of these objectives individually, it is possible to create utility functions (FC is the fraction of raw capacity used):

$$\begin{aligned}
\mathrm{U}_1\left(NINES_{DS}\right) &= \min\left(NINES_{DS}, 4\right) \\
\mathrm{U}_2\left(IOPS_{WL}\right) &= \min\left(IOPS_{WL}, 90\right) \\
\mathrm{U}_3\left(FC\right) &= 1 - FC
\end{aligned}$$

The general form of these priority-ranking scenarios is to have a series of n functions, with the first $n-1$ expressing specific targets and the final function being a maximization. The first two functions use the $\min(\cdot)$ function to limit the utility to the value when the objective is reached. This prevents the system from attempting to improve beyond the stated objectives.

The separate utility functions must still be combined into a single function. This can be done by scaling the individual functions, ensuring that improvements to a higher ranked metric take precedence over those of lower priority. For scaling the individual functions, an arbitrary scaling factor, S, will be used. It will also be assumed that the individual utility functions are bounded, with U_i^0 denoting the minimum value of the ith function and U_i^* denoting its maximum. Each function will be scaled to create a new function, $\mathrm{U}_i'(\cdot)$:

$$\mathrm{U}_i'(\cdot) = \left\lfloor (S-1)\,\frac{\mathrm{U}_i(\cdot) - U_i^0}{U_i^* - U_i^0} \right\rfloor$$

This scaling ensures that each $\mathrm{U}_i'(\cdot)$ covers the range of integers between 0 and $S-1$, inclusive. The use of integers quantizes the possible values for the metrics, making their precision explicit and ensuring that (small) fluctuations in a higher ranked metric cannot be offset by a large change to a

lower ranked metric. The full utility function can be constructed as:

$$\mathrm{U}(\cdot) = \sum_{i=0}^{n-1} S^{n-i-1} \mathrm{U}_i(\cdot)$$

For the example set of priorities above, the final utility function would be (assuming $S = 1000$):

$$\begin{aligned}\mathrm{U}(NINES_{DS}, IOPS_{WL}, FC) =& 1\mathrm{x}10^6 \left\lfloor \frac{999}{4} \mathrm{U}_1(NINES_{DS}) \right\rfloor \\ &+ 1000 \left\lfloor \frac{999}{90} \mathrm{U}_2(IOPS_{WL}) \right\rfloor \\ &+ \lfloor 999\mathrm{U}_3(FC) \rfloor\end{aligned}$$

The scale factor, S, can be chosen to be arbitrarily large, ensuring sufficient resolution of the component utility functions.

This scheme for priority-based utility has several limitations that stem from the lack of cardinality. The utility function does not intrinsically support making trade-offs. While useful for initial provisioning, the lack of support for making trade-offs precludes its use for data migration. However, an external method for making such trade-offs could be constructed and applied.

4.5 Utility with constraints

Real-world deployments frequently have constraints on the characteristics of the storage system that can be used. For example, there may be constraints on the physical space consumed by the storage system (e.g., square footage or rack units), on the total power consumption, on the system purchase cost, etc. Combining the techniques of the priority-based utility and the cost-based utility, functions can be created that perform much like the cost-based utility but adhere to applicable constraints.

The cost-based utility function, $\mathrm{U_c}(\cdot)$, can be created as described previously. The constraints can be expressed with the priority scheme above to create a second utility function, $\mathrm{U_p}(\cdot)$. The composite utility function could be expressed as:

$$\mathrm{U}(\cdot) = \begin{cases} \mathrm{U_c}(\cdot) & \text{if } \left(\mathrm{U_p}(\cdot) = U_p^*\right) \\ \mathrm{U_p}(\cdot) - U_p^* + U_c^0 & \text{otherwise} \end{cases}$$

where U_p^* is the maximum value of the priority-based function, and U_c^0 is a lower bound for the cost-based function. U_c^0 can be any value guaranteed to be less than $\mathrm{U_c}(\cdot)$ for reasonable configurations. This ensures that the utility value is equal to the cost-based value as long as the constraints are met, and if they are not met, the utility produced by the constraints will be less than all cost-based values.

When using this method for provisioning, verifying that the chosen configuration is greater than U_p^* ensures that the configuration satisfies all the constraints. Care must be taken when using this type of utility function for automatic tuning because trade-offs should be made only within the cost-based utility function (i.e., the value of utility should be greater than U_p^* at all times).

The above adjustments incorporate the constraints into the utility function, allowing a solver to satisfy the constraints in the normal course of maximizing utility. An alternate approach would be to specify the constraints directly and use a solution technique that is designed for constrained optimization.

4.6 Utility elicitation

Creating utility functions can be a difficult and time consuming task, particularly for a system administrator that is unfamiliar with utility and decision theory. Even using business costs, specifying utility is likely to be a challenge. Finding a way to help decision makers specify their preferences is a significant challenge in decision theory, not just with storage system administrators. Chen and Pu [2004] survey different utility elicitation methods and discuss systems that use them. Patrascu et al. [2005] describe a method for eliciting utility between automated systems, trading off the number of queries and fidelity of the optimization decision. While general solutions to the problem of eliciting utility functions are elusive, storage vendors already must address this problem (though in a more limited way) to guide provisioning using traditional methods. Adding a utility framework and the opportunity to incorporate business costs to the provisioning and tuning of storage will hopefully make this process easier.

Chapter 5

Provisioning solver

The previous two chapters described two of the three main components used for automatically provisioning storage. Section 1.2 first presented an overview of these components, and Figure 1.3 showed the components pictorially. Chapter 3 discussed the system models that predict the values of system metrics based on a storage system configuration, and Chapter 4 described the framework used to evaluate configurations based on those system metrics. When these two pieces (the models and utility function) are assembled, a storage system configuration is input, and an associated utility value describing the suitability of that configuration is output. This chapter will discuss the solver component that uses the feedback from the utility values to produce new storage configurations, closing this design loop.

This provisioning loop attempts to create a storage system design that best satisfies the objectives of a system administrator, as expressed by the administrator's utility function. The provisioning system is given a description of the system components, such as clients, workloads, datasets, and types of storage nodes. The objective is to determine how many and what type of storage nodes should be used and what data distribution should be chosen for each dataset. The final configuration may also include workload throttling that can be used to limit the I/O rate of less important workloads, allowing others to receive more system resources.

For this dissertation, several different approaches were used to create storage designs. When creating the solvers, every attempt was made to keep them independent from the other system components (i.e., the models and utility function). This choice was made because both the utility functions and system models are approximations of what would be expected in a real system. By keeping the solvers independent, it raises the likelihood that other models and utility functions could

be used without significantly affecting the solvers' effectiveness. The practical implication of this design choice is that the solvers know nothing about the internals of the models or utility function. They have no logic or rules about the likely affects of a configuration change. For example, the capacity consumption of a dataset is a simple function of the m and n encoding parameters (see Section 3.2), but the solvers do not have access to this information, requiring them to find space efficient configurations by trial-and-error. While this seems extreme in the case of capacity consumption, it is valuable for the metrics that are far less direct, such as performance.

When creating storage configurations, the goal is to create configurations that are *valid* and have a high utility value. For a configuration to be valid, it must be both *legal* and *feasible*. Legal configurations are those where the configuration is meaningful (i.e., the parameter values are consistent and all datasets have been placed in the design). For example, all legal configurations must maintain the proper relationship between m, n, and l (i.e., $1 \leq m \leq n \leq l$), and all datasets must be assigned in the storage system. As long as a configuration is legal, the system models and utility functions are able to analyze the configuration. The solvers must always produce legal configurations. This notion of "legal" configurations provides a baseline set of assumptions that ease the construction of storage system models. Before a configuration can be considered valid, it must feasible in addition to just legal. The feasibility of a configuration is related to whether it satisfies system constraints. Currently, the only feasibility constraint is capacity consumption. Every storage node must be able to store at least as much data as the configuration has assigned to it.

5.1 Exhaustive solver

The most obvious solution method to use is exhaustive search. The exhaustive solver generates every possible legal configuration, evaluates it, and remembers the valid configuration with the highest utility. Of the solvers created for this work, this is the only one that is guaranteed to find the optimal configuration. The price for that guarantee is an extremely long run time due to the large number of configurations that are analyzed.

The configurations are generated via nested loops. The outer-most loop controls l, searching from one to the total number of storage nodes allowed in the configuration, S. Inside this loop is another for n, running from one to l. The third loop controls m, between one and n. These three loops exhaustively generate values for the encoding parameters. Inside these, there is one final loop that controls which of the S storage nodes receive the l data fragments. This leads to $\binom{S}{l}$ choices in this inner-most loop. Each additional dataset that must be assigned causes this set of loops to

be nested an additional time, causing an exponential increase in the number of configurations. For storage designs that use all the same type of storage node (or have a limited number of types), this method generates and analyzes a large number of equivalent configurations, greatly decreasing its efficiency. One possible enhancement would be to generate configurations in a way that ensures uniqueness.

Based on the description above, the number of configurations analyzed by the exhaustive solver is (d is the number of datasets):

$$\begin{aligned}
&\left(\sum_{l=1}^{S}\sum_{n=1}^{l}\sum_{m=1}^{n}\frac{S!}{l!\,(S-l)!}\right)^{d} \\
&=\left(\sum_{l=1}^{S}\sum_{n=1}^{l}\frac{nS!}{l!\,(S-l)!}\right)^{d} \\
&=\left(\sum_{l=1}^{S}\frac{l\,(l+1)\,S!}{2l!\,(S-l)!}\right)^{d} \\
&=\left(\sum_{l=1}^{S}\frac{(l+1)\,S!}{2\,(l-1)!\,(S-l)!}\right)^{d}
\end{aligned}$$

For a storage system with two workloads and up to eight storage nodes, there are a total of 7.9 million possible configurations. Because of the large configuration space, the usefulness of this solver is limited to examining small configurations to evaluate other, more efficient, solution techniques.

5.2 Random solver

Instead of exhaustively exploring the configuration space, the random solver samples a number of random storage system designs, returning the best design encountered after a fixed number of valid designs have been analyzed. The random solver is one of the simplest solution techniques that could be employed. It makes no use of the feedback provided by the utility score other than to remember the best configuration found.

The effectiveness of this solver is highly influenced by the particular problem. Since the solver is randomly sampling the configuration space, the distribution of utility values across that space affects how good a solution it can be expected to find. For example, some problems have a large

number of solutions that are within a small interval of the optimal utility. In this case, the solver should perform well (i.e., produce a solution that is near optimal). Other problems have very few configurations that are close to optimal, so the chance that the random solver will find one is low.

This expectation could be quantified by examining the cumulative distribution function (CDF) of utility across all valid configurations for a given problem. If the goal is to produce a solution that is at least 90% of optimal, by examining the CDF at that fraction of optimal (i.e., $f = \text{CDF}(0.9opt)$), it is possible estimate the number of configurations that the solver would need to evaluate. From the definition of the CDF, f is the fraction of configurations that are more than this threshold from optimal, meaning there is probability f that a random configuration will fail to be at least 90% of optimal. After i iterations of the solver, the probability that none of the configurations analyzed are within this threshold is $p = f^i$. One would typically want to choose a small value for p such as 0.1 and solve for i. Therefore, if t is the tolerance from optimal, above, i can be calculated as:

$$i = \frac{\ln(p)}{\ln(\text{CDF}(t \cdot opt))}$$

While this expression can be used to gauge the sensitivity of the solver to the shape of the CDF and the desired tolerances, it is unlikely to be of use in practice because it requires both the optimal and the CDF of utility to be known. Both of these can only be generated through exhaustive search of the space, obviating the need to solve the problem again.

5.2.1 Generating configurations

The above discussion assumes the random configurations are sampled uniformly from the space of all valid configurations. Unfortunately, uniform sampling of the dataset encodings and locations is not trivial. There are two steps to the process. First, legal configurations need to be generated, and second, only the feasible subset of these can be used by the solver. The random solver addresses the second step by simply not counting an infeasible solution and drawing another random configuration. This lengthens the run time based on the fraction of legal configurations that are feasible, and can have a significant impact when this ratio is low, such as when the raw dataset size is close to the raw storage system capacity.

To illustrate the difficulty in generating legal configurations uniformly, consider a restricted example with one dataset and two total storage nodes. The dataset may use configurations from 1-of-1 declustered across 1 to 2-of-2 declustered across 2, which is a total of four different data

encodings. All of these encodings with the exception of 1-of-1 declustered across 1 have only one possible configuration, but the 1-of-1 declustered across 1 case has two, producing a total of five legal configurations that must be sampled uniformly. The naive method for generating random configurations would begin by selecting one of the encoding parameters uniformly (e.g., m), then constraining the choices for the others based on this result. Unfortunately, there is no ordering of m, n, and l that makes this approach suitable. Instead, the encoding parameters must be weighted based on their likelihood across the entire configuration space. For example, if choosing m first, there must be a probability of $\frac{4}{5}$ to choose $m = 1$ and $\frac{1}{5}$ to choose $m = 2$. Based on the outcome of the m choice, there are analogous weightings for n and finally l. Once the encoding parameters are chosen, l out of the S total storage nodes can be selected uniformly to hold the data fragments. The random solver pre-computes the tree of weights for m, n, and l at the beginning of each run, allowing the generation of individual configurations to execute quickly. In the evaluations of the solver, the magnitude of l is constrained to be less than or equal to eight, limiting the size of the lookup table. Eight was chosen for two reasons. First, eight is the limit of the exhaustive solver with two workloads to execute within a reasonable runtime. Second, in the Ursa Minor prototype, performance degrades when large numbers of storage nodes send data to a single client simultaneously, and eight exceeds this limit, making further predictions meaningless. This performance anomaly in Ursa Minor is due to the Incast Problem [Nagle et al., 2004].

5.3 Greedy solver

The number of configurations examined by the exhaustive solver is strongly influenced by the number of datasets that are being placed in the system. Specifically, the number of configurations is exponential in the number of datasets. The greedy solver is an attempt to constrain the number of configurations that are searched by optimizing the datasets individually instead of examining the "cross product" of all possible combinations. To this end, the greedy solver uses the exhaustive solver to optimize individual datasets, one at a time.

The greedy solver begins by generating a random order in which to optimize the individual datasets. This ordering then remains constant throughout the solver's execution. All datasets are initially placed randomly using the algorithms of the random solver. This initial random placement ensures a legal starting configuration for each dataset. This is required because all datasets must be placed before a configuration can be evaluated by the system models. Beginning with the first dataset, the greedy solver calls the exhaustive solver to find its optimal data distribution. The ex-

haustive search only changes the dataset currently being optimized, limiting the search for this step to

$$\sum_{l=1}^{S} \frac{(l+1)\,S!}{2\,(l-1)!\,(S-l)!}$$

possibilities. This first dataset is assigned to the optimal configuration generated by the search and the solver proceeds to the second dataset, optimizing it while holding all others (including the newly optimized first dataset) constant. This sequence proceeds until all datasets have been optimized once. Starting again from the beginning of the list, each dataset is optimized again. This continues until the solver is able to make a complete pass through all datasets without finding a new optimal configuration for any individual dataset. At this point, a local maximum has been reached, and the solver terminates.

This greedy approach to creating storage configurations optimizes each dataset individually instead of examining all possibilities simultaneously. This reduces the number of configurations searched by removing the exponential factor related to the number of datasets, instead trading it for a multiplicative factor (i.e., the d datasets are optimized sequentially). This changes the number of configurations that are examined to:

$$di\sum_{l=1}^{S} \frac{(l+1)\,S!}{2\,(l-1)!\,(S-l)!}$$

where d is the number of datasets and i is the number of iterations that occur before a local maximum is reached.

This algorithm is randomized based on the order in which the datasets are optimized and the initial random configuration that is chosen as the starting point. Changing either of these can lead to different locally optimal storage configurations.

5.4 Genetic solver

A genetic algorithm (GA) is an optimization technique that attempts to generate good solutions using techniques that are loosely based on natural selection and reproduction. Mitchell [1997] provides a general overview of genetic algorithms and discusses the main steps used for this type of optimization. The basic premise is that a number of potential solutions to the optimization problem form a *population*. There is some *fitness function* that is able to evaluate these candidate configurations, judging how well each candidate satisfies the problem's objective. Based on this evaluation,

candidates are chosen, using a *selection function*, for the next generation of solutions, biasing the selection toward the most fit candidates. These candidates are combined together using a *crossover operator* to mix their characteristics. They are also *mutated* to further explore the solution space. This process of evaluating, selecting, combining, and mutating continues through a number of generations of solutions until some stopping condition is reached. GAs have been applied successfully to many problems, including assigning portions of a database to different sites based on access patterns and network topologies [Corcoran and Hale, 1994] and for adjusting batch scheduling on a supercomputer to reduce wasted time [Feitelson and Naaman, 1999], among many others.

To create an effective GA, the functions and operators mentioned above must be defined to fit the details of the problem. For example, a fitness function must be created to evaluate configurations based on how well they solve the specific problem. The crossover operator should be able to structurally combine two different solutions, producing two new ones that are, ideally, both valid and exhibit a combination of the properties from their parents. The mutation operator needs to be able to make small changes to configurations in a manner that allows exploration of the configuration space while still producing valid configurations. The choice of how configurations are represented within the solver influence the ease with which these operations can be implemented.

5.4.1 Configuration representation

One of the first decisions when implementing a genetic algorithm is how to represent the potential configuration space. The goal is to use a compact representation that, based on its structure, ensures configurations are valid and provides access to the various configuration (tuning) parameters. The closer the representation is to meeting this ideal, the simpler (and more efficient) the crossover and mutation operations can be. In the general case, a configuration is represented as a fixed length vector of symbols, following the analogy that a chromosome is composed of a series of genes.

Instead of using a strict vector representation for the storage system configuration, it is more naturally encoded as a matrix. This matrix has one row for each dataset and one column for each storage node. Assuming binary values for each cell of the matrix, the representation naturally assigns a meaning to each position — the value at row d, column s controls whether dataset d is stored on storage node s. In effect, this matrix determines the location portion of each dataset's distribution. Unfortunately, it does nothing to represent the values of m and n. These values could be represented separately from this matrix (e.g., augmenting each row of the matrix with two integers, one for each), but this presents difficulty in ensuring that a configuration is legal. After each change

to the matrix, the solver would need to verify that $m \leq n$ and that n is less than or equal to the total number of "1's" in the corresponding matrix row. If a violation were found, it would need to be fixed because a configuration that is not legal cannot be evaluated. A better solution is to incorporate the values of m and n directly into the matrix, structurally ensuring the relationship between m, n, and l.

Instead of using just binary values, each location in the matrix may take on integer values between zero and three, inclusive. Just as above, a non-zero value at a particular location is used to indicate that a particular dataset (represented by the row) is stored on a specific storage node (represented by the column). The values of the m and n encoding parameters for a dataset are the number of entries in that dataset's row with values greater than two and one, respectively. For example, a row of [0 3 2 2 1] denotes an encoding of 1-of-3, with fragments stored on nodes two, three, four, and five ($l = 4$). This matrix representation was chosen because of the relative ease of maintaining the invariant: $m \leq n \leq l$. This works because incrementing m (placing a value of three in a row) also serves to increment n and assign a fragment location for the dataset. The only remaining condition that must be maintained to create legal configurations is that the value of m must be at least one. That is, there must be at least one "3" in each row of the matrix for the encoding to be valid. This condition can be verified relatively easily.

5.4.2 Fitness function

The fitness value determines how likely a candidate is to be selected for reproduction into the next generation. It accounts for both the utility of the candidate as well as the feasibility of the solution. Due to capacity constraints, not all configurations are feasible. To bias the solution toward feasible, high-utility solutions, the fitness value is:

$$fitness = \begin{cases} utility & \text{if}\,(utility \geq 2 \;\&\; OCC = 0) \\ \frac{1}{1+OCC} & \text{if}\,(OCC > 0) \\ \frac{1}{3-utility} + 1 & \text{otherwise} \end{cases}$$

where OCC is the sum of the over-committed capacity from the individual storage nodes (in MB). When the solution is feasible, $OCC = 0$ (no storage nodes are over-committed). For infeasible solutions, the fitness value will be between zero and one, with configurations having less over-committed capacity taking fitness values closer to one. This provides a bias toward less infeasible configurations. The third clause compresses utility values that are less than two (i.e., between two

and negative infinity) into the range of two to one. This ensures that feasible configurations always have higher fitness than infeasible configurations.

While this work has incorporated the feasibility of solutions into the fitness value, there are other approaches for encouraging feasible solutions. Chu and Beasley [1997] examined GAs for solving the Generalized Assignment Problem, and they added a specific step to the GA loop that was a domain-specific mutation that attempted to move candidates toward more feasible solutions. Another approach they used was to maintain "unfitness" values in addition to the standard fitness value. For single population GAs (where a single population is evolved as opposed to having discrete generations), the unfitness value can be used to control which candidates are replaced. Feltl and Raidl [2004] improved on these results using a penalty scheme similar to what is used above to handle capacity constraints.

5.4.3 Selection function

Using the fitness value for guidance, a selection function probabilistically chooses candidates to use as a basis for the next generation of solutions. There are two main techniques for using the fitness value to guide the selection. The first technique uses the magnitude of the fitness value as the weighting for selecting candidates. For example, if there is one candidate with a fitness value of 100 and another with fitness 50, the candidate with value 100 would be twice as likely to be selected as the one with a value of 50. This selection technique that is based on the relative magnitude of the fitness values is commonly known as Roulette selection. The second selection technique chooses candidates based on their rank order within the population, using the fitness values only to establish that ordering. One method of implementing this called Tournament Selection [Brindle, 1981]. In tournament selection, two candidates are chosen (uniformly) randomly from the population, and the algorithm returns the candidate with the higher fitness value.

The solver can use either Tournament or Roulette selection algorithms, but it defaults to Tournament. Empirically, the solver performs better using Tournament selection for utility functions that are ordinal such as the priority-based utility discussed in Section 4.4. The reason for this better performance is that once many of the objectives are satisfied, the magnitude of the utility value changes very little, causing Roulette selection to treat them nearly identically. Tournament selection is still able to strongly distinguish between them, allowing much more efficient convergence of the solver. Roulette selection has another disadvantage. Since it uses the magnitude of the fitness values, all

fitness values must be greater than zero. This was the motivation for using zero as the bound on fitness, above.

5.4.4 Crossover

The crossover operation combines two candidates from the current generation to produce two new candidates for the next generation. The intuition behind this step is to create new solutions that have properties from both of the original candidates (potentially achieving the best of both). There are several methods for performing the crossover operation, with the most common being single-point crossover. With this method, a random point is chosen along the vector representation of the candidate, and the alleles after this point are swapped between the two candidates. There are also variants that perform multi-point crossover. The method chosen for this work is *uniform crossover*. In uniform crossover, each allele has an independent 50% probability of coming from either original candidate. The algorithm used here is a slight variation, wherein, instead of treating each position in the configuration matrix as an independent position for crossover, it works at the level of a whole row at a time. In effect, this performs the crossover at the "dataset level" instead of on each location. This approach was chosen for two reasons. First, because it does not change the data distribution and only moves it between configurations, it cannot create an illegal configuration. Second, the individual values within a row have little meaning when taken independently. That is, the value of m and n depend on the whole row, not just a single value. For this reason, combining at the dataset level is more likely to form a new set of candidates with properties of the two original.

5.4.5 Mutation

The primary purpose of the mutation operator is to add randomness to the search. The algorithm used here operates on a single candidate at a time, changing a random location in the configuration matrix to a new, random value. This change can add or remove a dataset from a storage node if the value transitions to or from zero. It can also affect m, n, or both depending on the starting and ending values. Because it is changing the data encoding, the algorithm must ensure the resulting new encoding is valid (i.e., $m \geq 1$). Before a particular value is changed, it is verified that the current value, if it is a "3," is not the only "3" in that row. If a conflict is found, a different location is chosen for mutation. This does place some restriction on 1-of-n encodings, forcing them to transition to a $m = 2$ encoding before the storage node with the "3" can be vacated. In practice, this does not

appear to be a problem. An analogous check is done to limit the number of storage nodes per data distribution to some maximum value, typically eight.

In addition to this "single value" mutation, a mutation operator was created that chooses one of the datasets at random and completely randomized it (subject to the same constraints as above). This mutation operator did not perform well in the configurations tested as it caused too much of a change to the configuration.

5.4.6 Parameter tuning and stopping criteria

While GAs can be very effective for some optimization problems, they have a number of tuning parameters that can have an effect on their performance. Already mentioned were the choices of selection function, crossover operator, and mutation operator. Holding those constant, there are a number of other, lower-level, parameters to examine. First, the GA operates on a population of candidates. The size of this population must be chosen — large enough for a diverse set of alternatives to be examined, but large populations take more time to evaluate, constraining the number of generations that are examined per unit of time. The second parameter is the crossover probability. When a new population is created, some of this new population can be straight copies from the previous, and the rest have the crossover operator applied. The third parameter of interest is the fraction of candidates that are modified by the mutation operator.

When evaluating the choice for each of these parameters, there are two metrics of interest. First, how good is the solution that the algorithm produces? Various (improper) settings could cause the GA to not converge or become stuck at a local maximum. Second, how long does it take to produce a solution? Again, a poor choice of parameters could cause the GA to converge very slowly, taking an excessive amount of time to find a solution.

In order to judge these two criteria, it is necessary to determine some stopping criteria for the algorithm. Since there is, in general, no way to know when an optimal configuration is reached, the stopping condition is somewhat arbitrary. Common choices are to run for a fixed period of time, a fixed number of generations, or until the solver's progress slows below some threshold. For this evaluation, the latter is used. Specifically, if no new "best" configuration is found after a fixed number of generations since the last, the GA will terminate. The baseline configuration (before formal tuning) used a population size of 200. The stopping condition was chosen to be 20 generations of no additional progress. This equates to $200 \times 20 = 4000$ configurations being evaluated (with no additional progress).

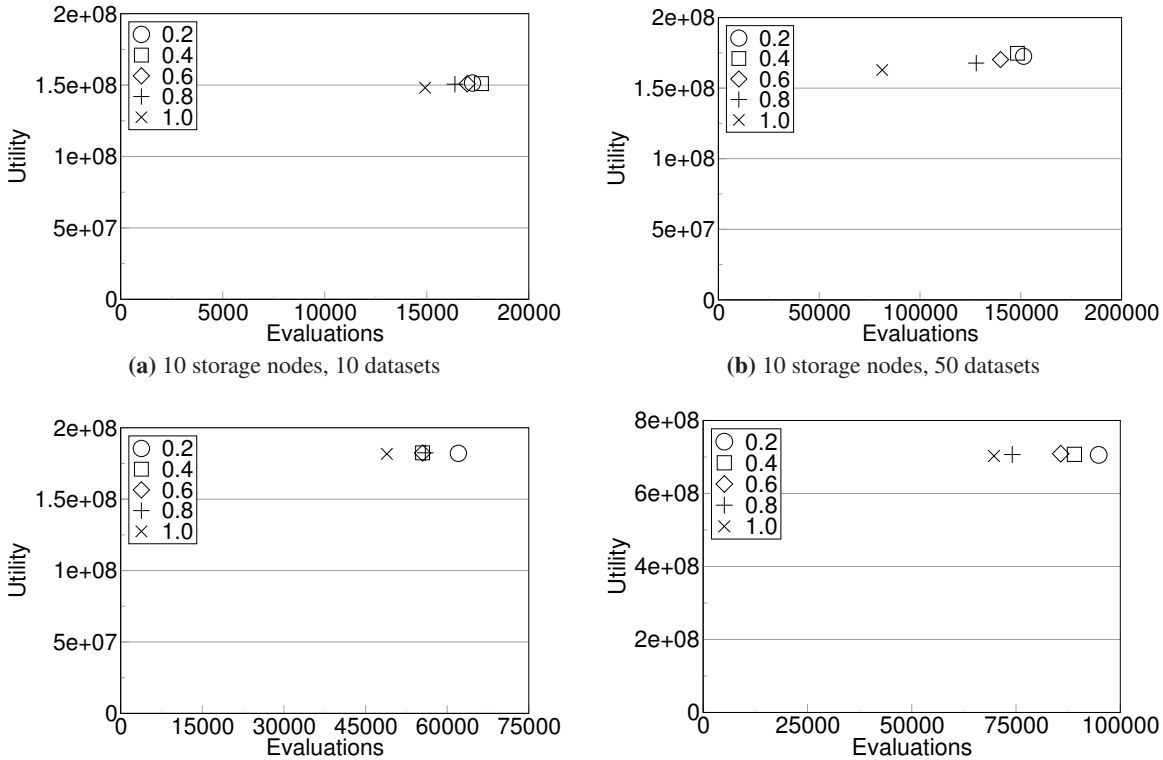
A sensitivity analysis was conducted for each of these three parameters (mutation probability, crossover probability, and population size) by evaluating the quality of the solutions produced (final utility and total evaluations to produce the solution) on four different problem sizes. The different scenarios are created by varying the number of datasets and storage nodes in the system configuration. The configuration consists of one "standard" client and one "6450" workload per dataset. The datasets are 5 GB each, and the storage node is of type "15k73." See Appendix A for the specifications of the clients, workloads, and storage nodes. The four problem sizes were created by using either 10 or 50 clients/workloads/datasets and 10 or 50 storage nodes. The utility function used for this evaluation is:

$$
\begin{aligned}
\mathrm{U}(\cdot) &= \sum_{WL,\,DS} \left(\mathrm{U}_{\mathrm{perf}}(\cdot) + \mathrm{U}_{\mathrm{av}}(\cdot) + \mathrm{U}_{\mathrm{rel}}(\cdot)\right) \\
\mathrm{U}_{\mathrm{perf}}(\cdot) &= (\$0.001)\, IOPS_{WL}\, AV_{DS} \left(\frac{3600\ \mathrm{s}}{\mathrm{hr}}\right)\left(\frac{24\ \mathrm{hr}}{\mathrm{day}}\right)\left(\frac{365.25\ \mathrm{day}}{\mathrm{yr}}\right) \\
\mathrm{U}_{\mathrm{av}}(\cdot) &= \left(\frac{-\$10,000}{\mathrm{hr}}\right)(1 - AV_{DS})\left(\frac{24\ \mathrm{hr}}{\mathrm{day}}\right)\left(\frac{365.25\ \mathrm{day}}{\mathrm{yr}}\right) \\
\mathrm{U}_{\mathrm{rel}}(\cdot) &= (-\$100\ \mathrm{M})\, AFR_{DS}
\end{aligned}
$$

Note that the utility functions for performance, availability, and reliability are summed across all the workloads/datasets in the particular scenario. Figure 5.1 presents the average of 10 trials with each of a 0.2, 0.4, 0.6, 0.8, and 1.0 probability of mutating a candidate configuration. The other parameters used were a probability of 1.0 for crossover and a population size of 200. Comparing the mutation rate based on the quality of the solution, it can be seen in the graphs that the quality of the solution differs by a maximum of seven percent. In most cases, it differs by much less. However, the time to create the solution differs by much more (as much as 85%). Based on these two observations, the final mutation rate was chosen to be 1.0, based solely on the time to create the solution.

The same set of scenarios were run again, varying the crossover rate between 0.2 and 1.0. This used a mutation rate of 1.0 and a population size of 200. As with the previous experiment, the quality of the solution differed relatively little based on the crossover rate, but the rate did influence the time to create the solution. The proper choice of crossover rate is not as clear cut as the mutation rate, but based on Figure 5.2, a crossover rate of 1.0 was chosen.

The final parameter evaluated was the size of the population. This experiment used population sizes of 25, 50, 100, 200, 400, 700, and 1000. The stopping condition for each parameter value was



(a) 10 storage nodes, 10 datasets

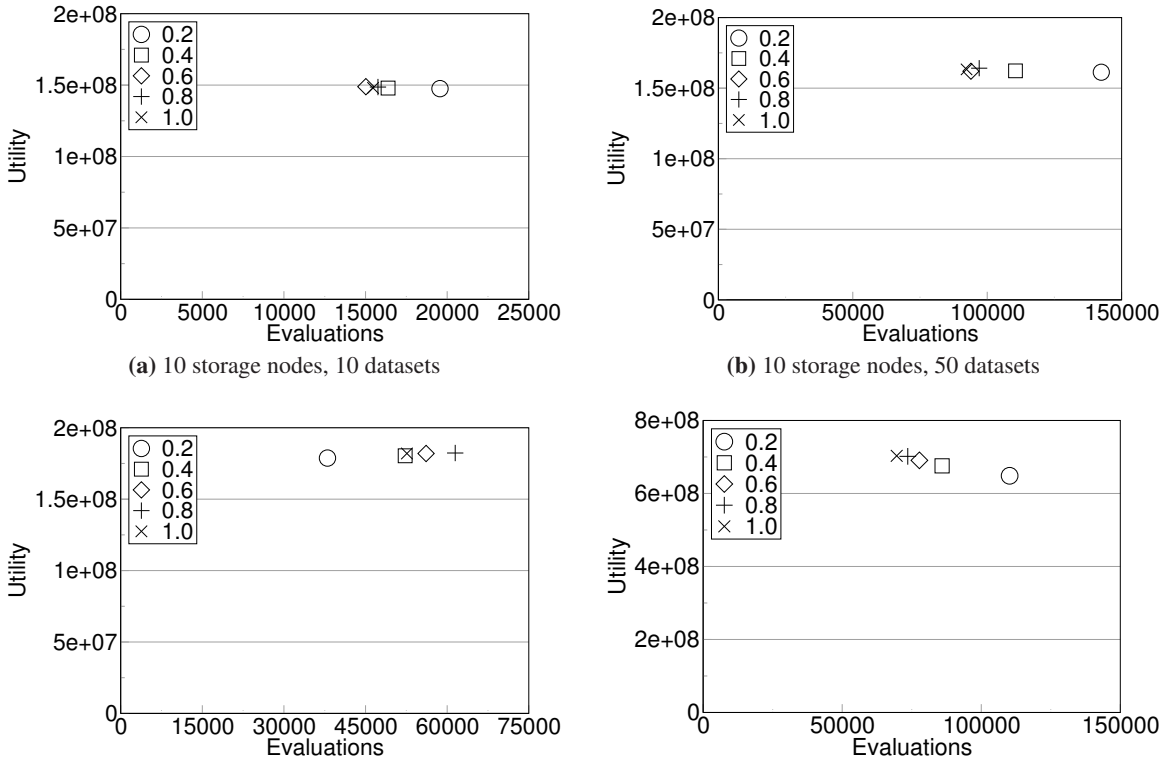
(b) 10 storage nodes, 50 datasets

(c) 50 storage nodes, 10 datasets

(d) 50 storage nodes, 50 datasets

Figure 5.1: Sensitivity to mutation probability – The four graphs show the utility of the final solution and the number of configurations evaluated (population size times total generations). Better tuning parameters lead to data points in the upper left of the graph (i.e., generating a better solution in less time). Each subgraph is for one of the four problem sizes, and each data point is labeled with the mutation probability used. All points are the average of ten trials.

adjusted to compensate for the different population sizes, keeping the number of candidates evaluated without making additional progress approximately constant. The formula used to determine the number of generations that would serve as the stopping condition was $G = \lceil 4000/p \rceil$, where p is the population size. A population size of 100 with a stopping criteria of 40 generations with no new best solution shows a clear benefit in solution time. As with the other parameters, it compares reasonably in solution quality. See Figure 5.3 for the specific results.



(a) 10 storage nodes, 10 datasets

(b) 10 storage nodes, 50 datasets

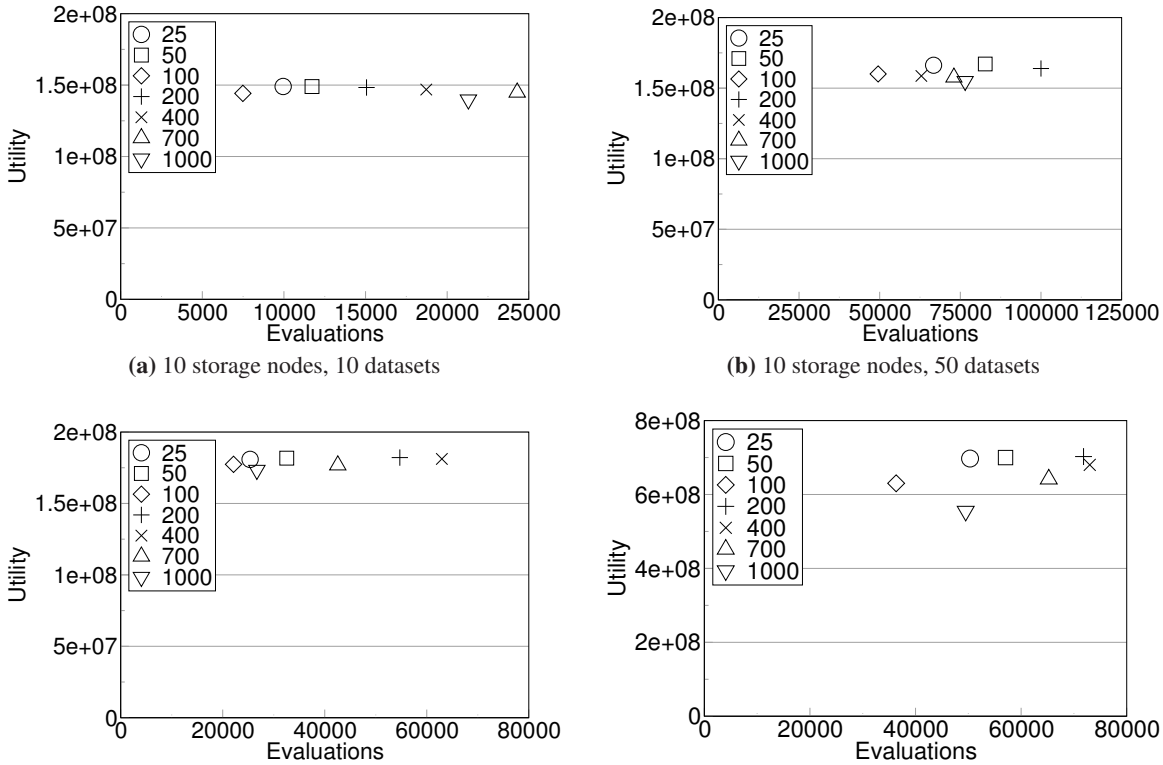
(c) 50 storage nodes, 10 datasets

(d) 50 storage nodes, 50 datasets

Figure 5.2: Sensitivity to crossover probability – The four graphs show the utility of the final solution and the number of configurations evaluated (population size times total generations). Better tuning parameters lead to data points in the upper left of the graph (i.e., generating a better solution in less time). Each subgraph is for one of the four problem sizes, and each data point is labeled with the crossover probability used. All points are the average of ten trials.

5.5 Comparison of solvers

Each of the solvers described above are positioned at different points in the design space of optimization techniques. The main axes for evaluation of these solvers are the quality of the solution that they produce and how long they require to produce that solution. At one extreme is the exhaustive solver. It is the only algorithm presented that is guaranteed to produce the optimal solution. The price paid for this guarantee is an impractical execution time for all but the smallest of problems.



(a) 10 storage nodes, 10 datasets

(b) 10 storage nodes, 50 datasets

(c) 50 storage nodes, 10 datasets

(d) 50 storage nodes, 50 datasets

Figure 5.3: Sensitivity to population size – The four graphs show the utility of the final solution and the number of configurations evaluated (population size times total generations). Better tuning parameters lead to data points in the upper left of the graph (i.e., generating a better solution in less time). Each subgraph is for one of the four problem sizes, and each data point is labeled with the population size used. All points are the average of ten trials.

The other algorithms sacrifice the guarantee of optimality to achieve better scalability. For example, the greedy solver optimizes each dataset independently, greatly improving the execution time, but the quality of the solution suffers significantly. The problem for the greedy solver arises because it is unable to move other workloads to make room for the one currently being optimized. This prevents resources from being effectively moved from one dataset to another, leading to suboptimal designs. Unfortunately, this design trade-off does not improve the execution time enough for it to be useful. While it scales approximately linearly with the number of datasets, it still has factorial

scaling with respect to the number of storage nodes.

The random and genetic solvers have the property that they can trade off runtime against the quality of the solution. For the random solver, the characteristics of the trade-off are determined by the distribution of utility values across all possible configurations, as discussed in Section 5.2. This dependence on the distribution of utility means it performs poorly for problems where there are relatively few solutions that are near optimal.

The genetic solver, which also has a significant random component, is much better at directing the search for solutions, making it perform well even when high quality solutions are very rare. To illustrate this, a scenario was created where the number of clients, datasets, and workloads are scaled together with a 1:1:1 ratio to control the problem's difficulty. The utility function is constructed so that it is possible to predict the number of "desirable" configurations as a fraction of the total number of possible storage configurations. The definition of "desirable" is that a dataset should have at least 4 "nines" of availability. Availability is used for this experiment because the data distribution is the sole determinant of its value, and one dataset's distribution does not affect another's availability. Using eight storage nodes of type "s500" and performing an exhaustive search with a single dataset, 464 of 2816 or 16.5% of the possible distributions meet the 4 nines criteria. By scaling the number of datasets in the scenario, solutions where all datasets have 4 nines of availability can be made an arbitrarily small fraction of the possible configurations. For example, with three datasets, $(464/2816)^3 = 0.4\%$ of the possible configurations have 4 nines for all three workloads.

The utility function for this scenario is:

$$U = \frac{1}{S} \cdot \sum_{i=1}^{S} \min\left(\text{NINES}\left(DS_i\right), 4\right)$$

S is the scale factor, corresponding to the number of datasets. This utility function will achieve its maximum value, four, when all datasets achieve at least 4 nines of availability. To ensure all possible data distributions are valid as the number of datasets are scaled, the size of the datasets relative to storage nodes are chosen to ensure the system is not capacity constrained.

Figure 5.4 shows how the GA solver performs as the number of datasets is scaled (from 5 up to 50). The graph plots the difficulty (the reciprocal of the fraction of configurations with 4 nines) of finding a 4 nines solution versus the number of configurations the solver evaluates before finding the first. It can be seen that exponential increases in the rarity of "good" solutions result in an approximately linear growth in the number of configurations that must be evaluated by the solver.

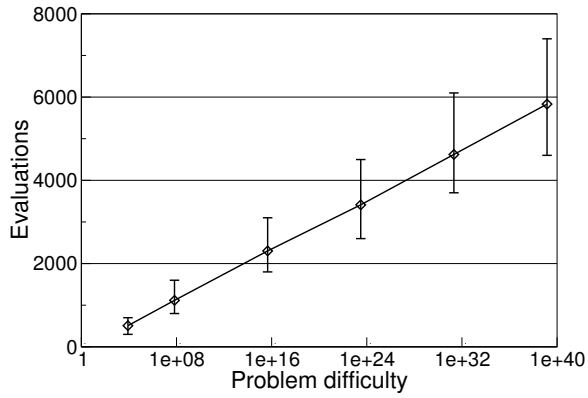


Figure 5.4: Finding rare solutions – This graph shows how the solution time changes as a function of how difficult the problem is to solve. The x-axis is the inverse probability that a random configuration is a valid solution ($\frac{1}{\Pr[\text{valid_solution}]}$). The graph shows the mean number (across 100 trials) of configurations that are evaluated by the GA solver before finding a valid solution. The error bars indicate the 5th and 95th percentiles.

Such linear growth (as opposed to exponential, or worse) can be explained by revisiting how the problem was constructed. The utility function was created such that the individual datasets are independent. The primary reason for this, as stated above, was to allow the number of "good" solutions to be easily calculated, but this independence also allows the optimization of the datasets to be performed independently (while the GA actually optimizes them all together, the analogy is reasonable). The resulting linear growth for this problem is quite encouraging, as the GA was able to automatically capitalize on the inherent independence of the datasets.

Another interpretation of this scaling graph is to consider it as a proxy for how well the GA solver converges toward the optimal configuration for this particular problem. Using the results of Figure 5.4 and examining the second data point, after just an average of 1016 evaluations, the algorithm has converged to the point where only one in 6.8×10^7 configurations would be as good or better. Likewise, for the final data point, in 5731 evaluations, it has converged to within one in 1.4×10^{39}. For the reasons cited above, the linear speed of convergence may not hold, but for this problem, the result is encouraging.

A more straightforward example of the convergence is shown in Figure 5.5. This experiment uses two clients, each of which has a single workload, and workloads each access their own dataset.

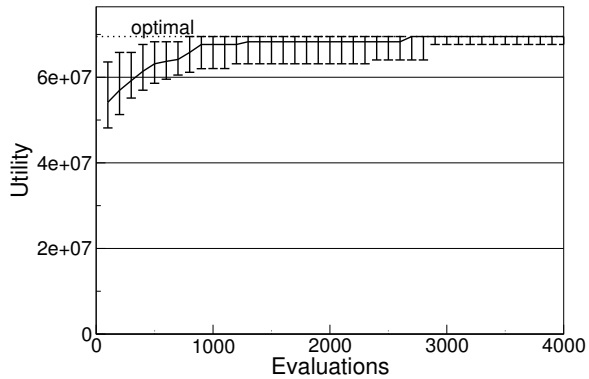


Figure 5.5: Convergence of the genetic solver – This graph shows how quickly the solver converges toward the optimal storage configuration. The line is the median over 100 trials, and the error bars indicate the 5th and 95th percentiles.

The datasets are constrained to encoding that use no more than $l = 8$ storage nodes. The storage nodes are of type "s500" from Table A.2, and the utility function is:

$$
\begin{aligned}
Utility &= U_{revenue} + U_{dataloss} + U_{downtime} \\
U_{revenue} &= \$0.001 \cdot AV_{DS} \cdot IOPS_{WL} \cdot \left(\frac{3.2\text{x}10^7 \text{ s}}{1 \text{ yr}} \right) \\
U_{dataloss} &= -\$100 \text{ M} \cdot AFR_{DS} \\
U_{downtime} &= \left(\frac{-\$10,000}{\text{hr}} \right) \cdot (1 - AV_{DS}) \cdot \left(\frac{8766 \text{ hr}}{1 \text{ yr}} \right)
\end{aligned}
$$

The exhaustive solution produces a utility of \$6.95x10^7/yr, using an encoding of 1-of-2 declustered across 4 storage nodes, and the two datasets are segregated onto their own set of 4 storage nodes. This optimal utility is shown as the dotted line near the top of the graph. The GA solver approaches this value quickly. Within five generations (500 total configurations evaluated), the median of 100 trials is within 10% of the optimum, and the bottom 5% achieves this level after just twelve generations (1200 total evaluations). Allowing for equivalent configurations, 3336 out of 7.9x10^6 total configurations are within 10% of the optimum. The utility values across the configuration space range from $-$\$7.23x10^7/yr to \$6.95x10^7/yr.

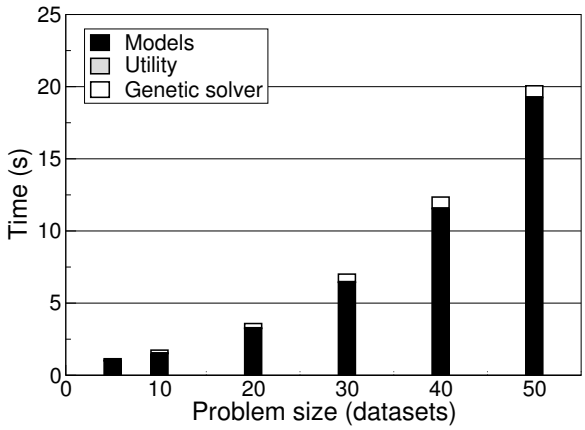


Figure 5.6: Genetic solver performance – This graph shows the breakdown of runtime between the models, utility function, and the genetic solver. All times are averages for evaluating a single population of 100 candidates. The time required to evaluate the utility function ranges from 7.6 ms to 45 ms for the 5 and 50 dataset sizes, respectively.

5.5.1 Genetic solver performance

The time required to generate a solution can be divided into three portions, corresponding to the three main parts of the provisioning loop, the models, the utility function, and the solver. Chapter 3 presented the performance for the models in isolation, and this section puts that performance data into context within the overall provisioning solver. Using the "4 nines" scenario above, the size of the system and the size of the utility function scales with the scaling factor S. Figure 5.6 shows the wall-clock time required per generation for the genetic solver. This time is then broken into components for the models, the utility calculation, and the overhead of the genetic solver, itself. The portion of the time spent evaluating the utility function is too small relative to the other components to be visible in the graph. These measurements were taken on a Dell PowerEdge 1855 with dual 3.40 GHz Intel Xeon CPUs and 3 GB of RAM running Linux kernel version 2.2.16 and Perl 5.8.8. The provisioning tool used only one of the two CPUs.

5.6 Overview of other potential optimization techniques

The genetic algorithm used for this work is just one of a number of potential optimization techniques. Other derivative-free techniques may also be appropriate. These techniques typically do not guarantee an optimal solution. Instead, they locate local maxima (or minima), and optionally use some sort of "restart" or "jump" to try to locate successively better solutions. While they do not guarantee an optimal solution, they have been applied successfully to a wide range of difficult to optimize problems. For example, the Disk Array Designer [Anderson et al., 2005] uses randomization to create an initial assignment then refines the configuration via several refinement operations that remove, randomize, then re-assign datasets.

For optimization problems of certain classes, more direct techniques can be used to find the optimal solution. For example, Linear Programming can be used for problems with linear constraints and objective functions. Other possibilities such as Integer Programming or Quadratic Programming have analogous restrictions on the problem formulation. Unfortunately, storage system models do not fit the restrictions of these techniques, making them inappropriate for use here.

5.7 Summary

Of the techniques evaluated in this work, the genetic solver has the best mix of solution quality and efficiency. It tends to produce better solutions than either the greedy or random solvers, and its running time is much better than the exhaustive approach. For the experiments in the following chapters, the exhaustive solver is used where practical, while the genetic solver is used on the larger problems.

Chapter 6

Provisioning and initial configuration

The first step of being able to control a distributed storage system is to be able to determine a suitable static configuration based on a description of the available components and the workloads/datasets to be served. This chapter puts together the components from the previous three (system models, utility, and an optimization technique) to evaluate several case studies of using utility for storage provisioning.

The first example shows how utility allows a structured approach to making trade-offs across different storage system metrics. The benefit to this structured approach is that it does not rely on intuition or rules-of-thumb that may or may not hold in a given environment. It allows trade-offs to be evaluated based on their merits in the particular environment.

The second example highlights a common problem with provisioning storage systems: handling external constraints. Being able to create optimal storage system configurations is a good starting point, but real-world environments can be constrained in many ways. This scenario shows how utility continues to provide guidance as external constraints force trade-offs to be made.

The third example illustrates how the cost of storage hardware can affect the chosen solution. Minimum cost provisioning systems are not affected by a uniform change in the cost of system components (i.e., they produce the same configuration). Utility, on the other hand, allows trade-offs that consider the purchase cost of the system hardware. This leads to situations where a price reduction (or increase) in hardware could change the optimal storage configuration because it changes the cost/benefit analysis of adding or removing components.

6.1 Trade-offs across metrics

Provisioning and configuration decisions affect multiple system metrics simultaneously; nearly every choice involves a trade-off. Using utility, it is possible to take a holistic view of the problem and make cost-effective choices.

Conventional wisdom suggests that "more important" datasets and workloads should be more available and reliable and that the associated storage system is likely to cost more to purchase and run. To evaluate this hypothesis, two scenarios were compared. Each scenario contained two identical workloads with corresponding datasets.

The utility functions used for the two scenarios are similar to the previous examples, having a penalty of \$10,000 per hour of downtime and \$100 M penalty for data loss. The cost to power the system is also added, at a rate of \$0.12 per kWh, and the purchase cost of the system (storage nodes) is amortized over a three year expected lifetime:

$$
\begin{aligned}
\mathrm{U}(\cdot) &= \mathrm{U}_{\text{perf}}(\cdot) + \mathrm{U}_{\text{av}}(\cdot) + \mathrm{U}_{\text{rel}}(\cdot) + \mathrm{U}_{\text{pwr}}(\cdot) + \mathrm{U}_{\text{purchase}}(\cdot) \\
\mathrm{U}_{\text{perf}}(IOPS_{WL}, AV_{DS}) &= (\text{see below}) \\
\mathrm{U}_{\text{avail}}(AV_{DS}) &= \left(\frac{-\$8.8\text{x}10^{7}}{\text{yr}}\right)(1 - AV_{DS}) \\
\mathrm{U}_{\text{rel}}(AFR_{DS}) &= (-\$100\text{ M})\,AFR_{DS} \\
\mathrm{U}_{\text{pwr}}(P) &= \left(\frac{-\$1.05}{\text{W}\cdot\text{yr}}\right)P \\
\mathrm{U}_{\text{purchase}}(C) &= \left(\frac{-C}{3\text{ yr}}\right)
\end{aligned}
$$

The two scenarios differ in the revenue they generate. The first generates 0.1¢ per I/O while the second only generates 0.01¢:

$$\begin{aligned}
\mathrm{U}_{\text{perf } 0.1}\left(IOPS_{WL}, AV_{DS}\right) &= (\$0.001)\, IOPS_{WL}\, AV_{DS} \left(\frac{3600 \text{ s}}{\text{hr}}\right)\left(\frac{24 \text{ hr}}{\text{day}}\right)\left(\frac{365.25 \text{ day}}{\text{yr}}\right) \\
&= \left(\frac{3.2\text{x}10^4 \text{ s}}{\text{yr}}\right) IOPS_{WL}\, AV_{DS} \\
\mathrm{U}_{\text{perf } 0.01}\left(IOPS_{WL}, AV_{DS}\right) &= (\$0.0001)\, IOPS_{WL}\, AV_{DS} \left(\frac{3600 \text{ s}}{\text{hr}}\right)\left(\frac{24 \text{ hr}}{\text{day}}\right)\left(\frac{365.25 \text{ day}}{\text{yr}}\right) \\
&= \left(\frac{3.2\text{x}10^3 \text{ s}}{\text{yr}}\right) IOPS_{WL}\, AV_{DS}
\end{aligned}$$

Based on this revenue difference, it would be easy to assume that the workload generating more revenue is "more important" than the other, requiring a higher (or at least the same) level of data protection. This assumption fails to account for the compromises necessary to achieve a particular level of availability.

Table 6.1 shows the results of provisioning these two systems. It shows both the metrics and costs for each part of the utility function. The table shows the optimal configuration for each scenario (as determined by the exhaustive solver): 1-of-2 declustered across 6 (1/2/6) for the 0.1¢ scenario and 1-of-3 declustered across 7 (1/3/7) for the 0.01¢ scenario. As a point of comparison, it also shows the results of using the other scenario's optimal configuration for each. Both scenarios used "s500" storage nodes and the "rrw" workload (See Table A.2 and Table A.3).

Examining the various contributions to the total utility, it can be seen that the main trade-off between these two scenarios is in performance versus availability. For the scenario with the higher revenue per I/O, it is advantageous to choose the data distribution with higher performance at the cost of lower availability (1/2/6) because the revenue generated by the extra $1250 - 1041 = 209$ I/Os per second per workload more than offsets the cost incurred by the extra downtime of this configuration. For the lower revenue scenario, the extra throughput cannot offset the availability difference, causing the lower performing, more available data distribution (1/3/7) to be preferred.

It is important to remember that these configurations are a balance of competing factors (mainly performance and availability in this case). Taking this trade-off to an extreme, such as choosing a very high performance data distribution with no regard to availability and reliability, results in poor utility. For example, using a 1/1/6 distribution for the 0.1¢ scenario provides only $33.7 M/yr in

Table 6.1: Effect of workload importance – A workload that generates more revenue per I/O should not necessarily have a higher level of data protection. This table compares two scenarios that differ only in the average revenue generated per completed I/O. The "more valuable" dataset's optimal data distribution is less available than that of the "less valuable" dataset because the cost of the additional downtime is more than offset by the additional performance of a less available data distribution. The data distributions in the table are written as: *m*/*n*/*l*.

	Metric values		**0.1¢ per I/O**		**0.01¢ per I/O**	
Distribution	1/2/6	1/3/7	1/2/6 (opt)	1/3/7	1/2/6	1/3/7 (opt)
Performance	1250 IO/s	1041 IO/s	\$76.3 M	\$65.4 M	\$7.6 M	\$6.5 M
Availability	1.5 nines	2.4 nines	−\$2.8 M	−\$329 k	−\$2.8 M	−\$329 k
Reliability	$2.0\text{x}10^{-6}$ afr	$1.1\text{x}10^{-10}$ afr	−\$407	−\$0.02	−\$407	−\$0.02
Power	600 W	700 W	−\$631	−\$756	−\$631	−\$756
Purchase cost	\$60,000	\$70,000	−\$20 k	−\$23 k	−\$20 k	−\$23 k
Total utility			**\$73.4 M**	\$65.1 M	\$4.7 M	**\$6.2 M**

utility because the reliability and availability costs now dominate.

Sacrificing availability for performance in a business scenario goes against the conventional wisdom of storage provisioning which typically places a strict ordering to availability and performance objectives. By using utility to analyze potential configurations, the multiple competing objectives can be examined analytically, providing evidence to explain and justify a particular storage solution.

6.2 Storage on a budget

Even with the ability to quantify the costs and benefits of a particular storage solution, it is not always possible for a system administrator to acquire the optimal system due to external constraints. These constraints may be due to limited power, cooling, or floor space within the data center, but the most common constraint is likely to be a limited storage budget. For example, the "optimal" storage system for a particular scenario may be too expensive for the system administrator to purchase with his limited budget. Presenting the administrator with this solution does him no good if he cannot afford it. Using utility, he has the ability to scale down this solution to find one that fits within his budget.

Table 6.2: Designing for limited budgets – The optimal system configuration costs a total of \$70 k, but the utility function can be used to choose the best configuration that fits within other (arbitrary) budgets as well. Limiting the budget constrains the total hardware available, and the utility function guides the solution to make cost effective trade-offs as the system capabilities are scaled down to meet the limited budget. Note the the storage "budget" in the table is the total cost of the system, while the "amortized purchase cost" reflects this amount spread over the system's expected three year lifetime.

Budget	**\$70 k**	**\$30 k**	**\$20 k**
Distribution	1/3/7	1/2/3	1/2/2
Performance	\$6.5 M	\$6.7 M	\$5.8 M
Availability	−\$329 k	−\$636 k	−\$219 k
Reliability	−\$0.02	−\$163	−\$81
Power	−\$736	−\$316	−\$210
Amortized purchase cost	−\$23 k	−\$10 k	−\$6.6 k
Total utility	\$6.2 M/yr	\$6.1 M/yr	\$5.6 M/yr

Using the 0.01¢ scenario from above as an example, the optimal solution uses fourteen storage nodes (seven for each dataset) and costs \$70 k. For an administrator whose budget cannot accommodate this purchase, this solution is unworkable. Table 6.2 compares this optimal solution to two alternatives that have constraints on the total cost of the storage hardware. The first alternative has a limit of \$30 k on the purchase cost, and the second further reduces the budget to \$20 k. Notice that these two alternatives use six and four storage nodes respectively (at \$5000 each) to stay within their budget. The "purchase cost" in the table reflects this cost spread over the system's expected three year lifetime.

With each reduction in budget, the total system utility decreases as expected, but the chosen configuration at each level still balances the relevant system metrics to maximize utility as much as possible. The reduction from optimal (\$70 k) to \$30 k results in choosing a configuration that sacrifices some availability to gain performance, resulting in only a 2% loss of overall utility. The reduction to \$20 k from optimal leads to a 10% loss of utility as performance is significantly reduced.

As this example illustrates, utility presents the opportunity to make trade-offs even among non-optimal or in less than ideal situations. This ability to account for external constraints makes utility-based tools more helpful than those that perform only minimum cost provisioning by allowing solutions to be identified that conform to real-world constraints.

6.3 Effects of hardware cost

Even without budgetary constraints, the price of the storage hardware can affect the proper solution. Consider the trace processing example from Section 4.3.1, which is based on the activities of graduate students processing file system traces. The utility function, developed earlier, is reproduced here:

$$
\begin{aligned}
\mathrm{U}_{\mathrm{perf}}\left(BW_{WL}\right) &= \left(\frac{-\$6.72\mathrm{x}10^{4}\ \mathrm{MB}}{\mathrm{yr{\cdot}s}}\right)\left(\frac{1}{BW_{WL}}\right) \\
\mathrm{U}_{\mathrm{av}}\left(AV_{DS}\right) &= \left(\frac{-\$3.94\mathrm{x}10^{5}}{\mathrm{yr}}\right)\left(1-AV_{DS}\right) \\
\mathrm{U}_{\mathrm{rel}}\left(AFR_{DS}\right) &= \left(-\$525\right)AFR_{DS} \\
\mathrm{U}_{\mathrm{pwr}}\left(P\right) &= \left(\frac{-\$1.05}{\mathrm{W{\cdot}yr}}\right)P \\
\mathrm{U}_{\mathrm{purchase}}\left(C\right) &= \left(\frac{-C}{3\ \mathrm{yr}}\right)
\end{aligned}
$$

The workload is modeled using "sread" from Table A.3. Provisioning this system using storage nodes "s500e" as described in Table A.2, with a cost of $10 k per node, leads to a solution using two storage nodes and 2-way mirroring. Table 6.3 shows a breakdown of the costs using these "expensive" storage nodes. If the cost of a storage node is reduced to $2 k each (type "s500c"), the total costs obviously decrease due to the lower acquisition cost (second column of Table 6.3). More interestingly, the optimal configuration for the system also changes, because it is now cost effective to purchase an additional storage node. By comparing the last two columns in the table, the additional annualized cost of the third storage node ($667) is more than offset by the increase in availability that it can contribute ($1457). In fact, this new configuration provides almost a 20% reduction in annual costs.

6.4 Effects of model error

While researchers strive for accurate models, it is highly likely that some error will persist. This modeling error has the potential to cause an automated provisioning system to choose suboptimal storage configurations.

Table 6.3: Price affects the optimal configuration – The optimal configuration using "expensive" ($10 k each) storage nodes is two-way mirroring, but if the cost of the storage nodes is reduced to $2000, it is now advantageous to maintain an additional replica of the data to increase availability. This additional storage node results in an almost 20% decrease in expected yearly costs for the storage system.

	Expensive (opt)	**Cheap (same)**	**Cheap (opt)**
Distribution	1/2/2	1/2/2	1/3/3
Performance	−$874	−$874	−$862
Availability	−$1534	−$1534	−$77
Reliability	−$0	−$0	−$0
Power	−$105	−$105	−$158
Purchase cost	−$6667	−$1333	−$2000
Total utility	−$9180/yr	−$3846/yr	−$3097/yr

In this experiment, error was intentionally introduced into the performance model, and the effect on utility was examined. The 0.01¢ scenario from Section 6.1 was used as the basis for the comparison. It was shown previously that the optimal configuration produces a utility of $\$6.19\text{x}10^6$ per year. This experiment introduces a random, uniformly chosen, error between 0% and some maximum into each metric produced by the performance model. Each metric is adjusted by a unique error percentage, and each unique system configuration is adjusted with a different set of random errors. However, for each trial, the error in a given configuration's metrics is held constant. These perturbed metrics are used by the utility function and solver to generate a system configuration. Once the solver produces a final configuration, that configuration's true utility is calculated using the system models without introducing error. Comparing this actual utility with the utility produced by the solver when there is no modeling error can be used to gauge the impact of the modeling error.

Table 6.4 shows the average (true) utility produced across 100 trials for errors ranging from 10% to 40%. The mean error for each of these experiments is the percentage by which the mean utility is less than the optimal utility. In this scenario, these errors are considerably less than the modeling error. The first line of the table shows the results without introducing any modeling error, and it demonstrates that the genetic solver was able to find the optimal in each of the 100 trials (i.e., the mean utility is the same as the minimum utility).

Table 6.4: Effect of model error – Error in the system models can cause the provisioning solver to create sub-optimal solutions. This table shows the effect of varying amounts of error on the utility of the final configuration. Between 0% and 40% error was introduced into the performance model for the 0.01¢ scenario from Section 6.1. The mean values are based on 100 trials.

Model error	Mean utility	Mean decrease in utility
0%	\$6.19x10^6/yr	–
10%	\$6.15x10^6/yr	0.6%
20%	\$6.08x10^6/yr	1.8%
30%	\$5.91x10^6/yr	4.5%
40%	\$5.78x10^6/yr	6.5%

Chapter 7

Migration solver

Previous chapters have shown how utility can be used to guide the initial provisioning of a cluster-based storage system. However, once a system is deployed, the workloads and hardware, as well as the administrator's objectives, can change. As a result of these changes, the utility provided by that initial configuration may decrease, and other configurations may better meet the administrator's objectives. The challenge is to determine whether it is advantageous to reconfigure (i.e., change the data encodings and locations), evaluating new configurations and migration plans by balancing the cost of reconfiguration against the potential benefit.

There has been considerable work in dynamically tuning systems to achieve a set of performance goals. For example, Parekh et al. [2002] used control theory to maintain performance SLAs for clients in Lotus Notes. Their work adjusted various configuration parameters to achieve a desired response time for clients. Karlsson et al. [2004] used control theory to divide throughput between clients according to a fixed schedule while maintaining latency targets, and Uttamchandani et al. [2005] attempted to meet performance SLAs by throttling clients based on calculations from support vector machines. Directly related to the reconfiguration of interest in this work is Aqueduct [Lu et al., 2002], which controlled the speed of data migration with the objective of minimizing the migration time while not violating performance requirements for foreground workloads. The majority of the related work uses fixed performance targets and some type of control system to match it. Just as with provisioning, this dissertation does not attempt to provide a fixed level of particular system metrics. Instead, it attempts to choose new configurations and suitable migration plans to maximize the utility that the system provides.

7.1 Trade-offs involved in migration

The potential benefit of migrating data is clear — moving to a new configuration that provides higher utility. However, much like initial provisioning, there are many trade-offs to be considered. Choosing to migrate data has an associated cost for several system metrics, including performance and data protection metrics. Performing a data migration quickly impacts foreground performance more than performing it slowly, but the benefits of the new configuration are achieved sooner. Additionally, during data migration within Ursa Minor, the availability and reliability of the dataset relies on both the new and old data distributions, hurting these metrics for the duration of the change (and arguing for a fast data migration).

Beyond just the choice of how fast to migrate data, the cost of the migration can also influence the choice of final configuration. For example, there may be a new, potentially optimal, configuration that requires all datasets to be moved. This choice must be weighed against less optimal ending configurations that require less disruptive data migrations. All of these factors need to be combined in a rigorous manner to permit a storage system to automatically generate and implement an optimization plan.

Currently, system administrators are ill-equipped to thoroughly evaluate data migration trade-offs. This leads to "best practices" such as always performing migration slowly and during non-peak times. In addition, administrators are very conservative with migration decisions, favoring no action unless there is a demonstrated need for a change. Unfortunately, without a disciplined approach to evaluating the trade-offs, even the notion of a "demonstrated need" is subjective!

7.2 Valuing a migration plan

Looking back at the provisioning problem, the utility values of candidate configurations could be directly compared because the hardware and workloads were assumed to be static, producing a constant rate of utility. For data migration, however, the system is dynamic. Utility changes throughout the operation, and two plans may take different amounts of time to reach their final configuration. Additionally, the final configuration may be different for each of the plans. To compare such disparate plans, it is necessary to consider the utility that each produce over time.

Following the approach of using business costs to evaluate storage configurations and taking inspiration from economics, the expected stream of future utility can be discounted back to a *Net Present Value*. This allows alternatives to be compared based on an equivalent present utility value.

In effect, the predicted future utility from the migration and final configuration is viewed as a stream of income that is discounted at some appropriate rate. This is analogous to how an investment choice (e.g., choosing whether to invest in a company) might be weighed based on an analysis of discounted cash flow.

Up to this point, utility has been described as a function of constant system metrics (e.g., $\mathrm{U}(BW_{WL},\ldots)$). While it is still assumed that the utility function remains constant, the metric values on which it is based do change, causing the rate of utility to change with time. Instead of writing this as $\mathrm{U}(\mathrm{BW_{WL}}(t))$, we will continue to refer to utility functions with the old notation and the understanding that the system metrics may change with time. The time-dependent stream of utility that is produced will be written as $\mathrm{U}(t)$.

Typically, the present value of an income stream (e.g., for a discounted cash flow analysis) is modeled as a series of payments at discrete times. The utility function, however, is modeled as a continuous stream where the value of the utility function at any point in time is the instantaneous rate at which utility is being acquired. This leads to a discounting of that rate in a manner similar to continuously compounding interest, wherein the value is discounted by an exponential factor:

$$\mathrm{U_{PV}}(t) = \mathrm{U}(t)\,\mathrm{e}^{-rt}$$

This (discounted) rate can then be integrated over the lifetime of the stream of utility to produce the present value. This present value is a value for utility (the expected accumulation over the system's lifetime), not a rate. Assuming a data migration begins at $t = 0$ and the system will continue to run for the foreseeable future, the calculation is:

$$\begin{aligned} U_{PV} &= \int_0^\infty \mathrm{U_{PV}}(t)\,dt \\ &= \int_0^\infty \mathrm{U}(t)\,\mathrm{e}^{-rt}dt \end{aligned}$$

It is important to note that the integral extends to infinity, meaning that the ending configuration of the migration is also included in this calculation. The actual migration operation begins at $t = 0$ and continues for T_m seconds. For times $t > T_m$, the system is in the final configuration, and the value of the utility function is assumed to be constant, having the value associated with this final

configuration.[1]

The discount rate, r, adjusts the weight placed on short-term utility versus long-term benefit from a migration. Higher values of r lead to a faster decay, more heavily discounting future utility and increasing the relative importance of short-term utility. An interesting area for future work is exploring how to accurately determine this discount rate. The approach used here is to view this discount factor as accounting for the risk that the storage system (or workloads) will change, invalidating the prediction of utility. This interpretation is consistent with the formula above, assuming an exponential distribution in the frequency of system changes. That is, the probability that the system will not have changed between $t = 0$ and $t = T$ would be e^{-rT}, and the mean time between changes would be $1/r$. This implies that for a storage system that we expect to change, on average, once per week,

$$\begin{aligned}\frac{1}{r} &= 1 \text{ wk}\left(\frac{7 \text{ day}}{\text{wk}}\right)\left(\frac{24 \text{ hr}}{\text{day}}\right)\left(\frac{3600 \text{ s}}{\text{hr}}\right)\\ r &= 1.65\text{x}10^{-6} \text{ s}^{-1}\end{aligned}$$

This discount rate could be considered a lower bound on the actual discount rate, because it only accounts for the risk of change to the system. Typically, interest rates incorporate both a component for the risk and an additional amount, known as the risk-free rate, which can be viewed as the extra reward for the delay in receiving income. Because of the difference in magnitude between the risk and risk-free rates (the risk-free rate is typically only several percent per year), the risk-free component will not be considered further, and the discount rate will only use the mean time to change as shown above.

One might be temped to consider an alternate approach for calculating the present value by using a more "typical" discount rate (e.g., 10% annually) and limiting the upper bound of the integral to constrain the impact of the uncertain future. Returning to the example of expecting the storage system to change once per week (on average), it may be tempting to change the present value integral to be: $\int_0^{1\text{wk}} \mathrm{U}(t)\,\mathrm{e}^{-rt}dt$, with the discount rate as $r = 10\%\ \text{yr}^{-1} = 3.17\text{x}10^{-9}\ \text{s}^{-1}$. There are two problems associated with this line of reasoning. First, by limiting the upper bound of the integral, the calculation disregards utility beyond this threshold (1 wk) into the future. This might be plausible if the storage system were to change exactly once per week, but the intent is to

[1] In the case where there is no migration (i.e., $\mathrm{U}(t)$ is a constant) this calculation produces the same ordering of configurations as directly comparing the (constant) utility values. Thus, the discounting used for migration is consistent with the direct comparisons that were used for provisioning. See Section 7.2.1 for the specific formula.

model a probabilistic change (e.g., an *average* of once per week), leading to erroneous results with this approach. Further, in situations where a reconfiguration requires longer than this threshold to complete, the end configuration would not be incorporated into the results. This oversight could lead to short-term gains that produce poor configurations farther in the future. Second, the motivation to use "typical" discount rates comes from the desire to draw a parallel between the values seen in financial markets and those that should be applied to a storage system. Equating these two rates amounts to equating the risk as well. Using the same discount rate for valuing a company and a storage migration plan implies that the risk of bankruptcy for the company is equivalent to the risk of the storage system (or its workloads) changing. These two likelihoods are typically not of the same magnitude.

7.2.1 Comparison against the current configuration

Using the above method, different migration alternatives can be compared, regardless of how much data they move or what their ending configuration is. This method presents a way to identify attractive plans for reconfiguration, but it still leaves an open question of whether a given plan is worth implementing or the current configuration should be retained. In fact, the same methodology used for valuing a migration plan can be used to assign a value for retaining the current configuration (i.e., doing nothing). The insight is that keeping the configuration static is, in effect, a data migration that does not consume any time with an ending configuration that is the same as the initial configuration. Using this insight, the present value of keeping the current configuration is just:

$$\begin{aligned} U_{PV} &= \int_0^\infty \mathrm{U}(t)\,\mathrm{e}^{-rt}\,dt \\ &= \int_0^\infty U_{current}\,\mathrm{e}^{-rt}\,dt \\ &= U_{current}\int_0^\infty \mathrm{e}^{-rt}\,dt \\ &= \frac{U_{current}}{r} \end{aligned}$$

7.3 Modeling migration

Migration within the Ursa Minor architecture proceeds in two phases. First, the metadata service installs back-pointers at the new data location. These back-pointers contain metadata information sufficient to redirect clients, allowing them to read the data from its old (current) location. In the

second phase, all clients are redirected to the new data location, and a "migration coordinator" is started. By immediately redirecting the foreground workloads, writes will help move data into the new location, while reads will require an additional round trip to storage nodes (to follow the back-pointers) if the requested block has not yet been moved. The migration coordinator is the main data mover for migrations. It resembles an additional client and workload in the system, reading data from the old location and writing it to the new. The principal method for controlling the speed of data migration is by varying the rate at which the coordinator reads and writes data blocks.

The installation of the back-pointers by the metadata service was not modeled, because the operation is not in the critical path of servicing I/O requests. The migration coordinator was modeled as an additional client with one workload that accessed the dataset being moved. The choice to create a new client was somewhat arbitrary as the migration could also be performed by one of the existing clients. Modeling the datasets and workloads during migration required changes to the models that were presented in Chapter 3.

Availability and reliability The use of back-pointers for data migration has the disadvantage that both of the data distributions are critical to serve the dataset. Once a client makes its first write to the new location, the old data distribution no longer contains an up-to-date copy. Likewise, until all data has been copied, the old distribution is needed to fill the gaps that still contain back-pointers. This reliance on both data distributions has implications for both the availability and reliability of the dataset.

Both the (binomial) availability model and the reliability model treat the two data distributions as independent. The availability of the dataset is calculated as:

$$AV_{DS} = AV_{DD_{old}} \cdot AV_{DD_{new}}$$

The reliability is calculated by summing the annual failure rates of the distributions:

$$AFR_{DS} = AFR_{DD_{old}} + AFR_{DD_{new}}$$

Capacity During the data migration, both data distributions contain data and occupy space on the storage nodes. The capacity calculations (consumption, utilization, and blowup) are based on the full capacity being consumed by both distributions. This corresponds closely to the Ursa Minor prototype, where back-pointers are stored for each data block. Potential enhancements to Ursa

Minor include only consuming storage in the new data distribution when it is written and freeing storage in the old distribution once a data block has been copied (or overwritten) in the new location.

Performance The performance models are affected in two main areas, where the I/O requests are directed and how demands are calculated at storage nodes. To reuse the terminology from the description of the performance model, it affects the calculation of both the *visit ratio* and the *service time*. First, at different stages of the migration, there will be different load placed on each of the data distributions. When the migration begins, all I/O requests are sent to the new data distribution, and all reads will additionally be sent to the old distribution (because they are redirected by a back-pointer). As the migration progresses, more data will be stored in the new distribution, reducing the fraction of reads that are redirected. At the end of the migration, none of the reads will be sent to the old distribution because all of the back-pointers will have been replaced by data. To simplify the calculations, the model assumes that all I/O operations are spread uniformly across the address space of the data object. Using this assumption, if the visit ratio for the workload based on the individual data distributions are V_{read} and V_{write}, the old distribution's visit ratios become:

$$V'_{read} = (1-c)\,V_{read}$$
$$V'_{write} = 0$$

where c is the fraction of the dataset that has been migrated. All writes are directed solely to the new data distribution. The new data distribution's visit ratios are calculated normally.

The visit ratio for the workload of the migration coordinator is calculated normally, except read operations are directed to the old location, writes to the new, and the workload itself is modeled as a sequential access pattern, regardless of the foreground workloads.

The service time calculation is altered slightly. As described previously, the disk service time is affected by the number of workloads that access the disk. The reason for this parameter is to account for interference between workloads that would decrease the effective sequentiality. In keeping with this reasoning, the accesses directed at the old and new data distributions are counted as separate "workloads" in this formula even though they originate from a single foreground workload. The rationale for this treatment is that accessing different data distributions that share the same disk would incur a seek.

Power Since both data distributions are active during the data migration, all storage nodes for both distributions are assumed to be contributing to the power consumption of the storage system.

Given the above adjustments to the models, the remaining task is to model the actual progress of the data migration. The models and utility function are designed to assess static snapshots of the system configuration. This leads to an approach of quantizing the data migration into a series of steps and calculating the metrics and utility once for each step as the migration proceeds. The migration process could be quantized by either time (e.g., every 30 s) or by the fraction of data that has been migrated (e.g., every 1%). For simplicity, the latter was chosen. This provides a known value for c in the calculations above. Each of these migration quantum last for a varying amount of time, based on the rate of data migration. For example, it is known that for a dataset of B blocks, each quantum of size q (a fraction between 0 and 1) will require qB blocks to be migrated from the old data distribution to the new. This required number of blocks is moved by the combination of the migration coordinator and the writes of the foreground workload.

The migration coordinator reads from the old distribution and writes to the new, performing two I/O operations for each block moved. Assuming it does not re-copy any blocks that have already been written by the foreground, it contributes a consistent rate of $IOPS_{COORD}/2$ to the migration of the dataset currently being moved. [2] The foreground writes, on the other hand, contribute a varying amount based on the current fraction complete. Again, assuming that the foreground writes are spread uniformly, the total migration rate is (in blocks per second):

$$\mathrm{BPS}\,(c) = \frac{IOPS_{COORD}}{2} + (1-c) \cdot FracWrite \cdot IOPS_{FG}$$

Using this formula, the duration of the quantum can be calculated as:

$$\mathrm{Q_d}\,(c) = \frac{q \cdot B}{\mathrm{BPS}\,(c)}$$

[2] The coordinator can periodically request a list of blocks still containing back-pointers, reducing the number of re-copied blocks to an insignificant fraction.

The NPV of the stream of utility can then be calculated as a sum of the discounted quanta plus the discounted value of the end configuration:

$$U_{PV} = \sum_{i=0}^{1/q-1} \mathrm{Q_d}\left(iq\right) \cdot \mathrm{U}\left(iq\right) \cdot \mathrm{e}^{-r\mathrm{T_s}(iq)} + \int_{\mathrm{T_s}(1)}^{\infty} U_{final}\, \mathrm{e}^{-rt} dt$$

$$\mathrm{T_s}\left(c\right) = \sum_{i=0}^{c/q-1} \mathrm{Q_d}\left(iq\right)$$

$\mathrm{T_s}\left(c\right)$ is the time, in seconds, of the beginning of the quantum that models the migration step between the fractions c and $c+q$ complete.

7.4 Searching for good plans

There has been significant work on developing data migration plans. For example, Anderson et al. [2001a] compare several algorithms that move fixed-size data chunks, accounting for space constraints and attempting to complete the migration in as few steps as possible. A given storage device was only permitted to move one data chunk (either to send or receive) in each step. They chose the migration duration (in number of steps) as the main comparison criterion for the algorithms.

Given that others have developed methods to handle space constraints, that work will not be duplicated here. Those results could be used to augment the ordering of dataset migration or to create intermediate steps that would account for capacity limitations. The main goal of this portion of the dissertation is to value migration plans (and the resulting configuration) based on guidance from a utility function. The main parameters available for optimization are the ending configuration, the ordering of dataset movement, and the speed with which data is moved. The optimization system moves a single dataset at a time, and the speed is controlled by adjusting the multiprogramming level (number of outstanding requests) of the migration coordinator's workload.

Migration rate for a single dataset As an initial step toward creating a migration plan between two arbitrary configurations, this section only considers optimizing the migration speed between two fixed configurations that differ by exactly one data distribution. In this scenario, the single dataset to move is given. The challenge is to determine, for each migration quantum, the proper speed of the migration coordinator.

If the speed of the migration is also quantized (e.g., the multiprogramming level of the coordinator can have values between one and ten, or a total of $M = 10$ values), this implies a total of $M^{1/q}$ possible migration schedules. The insight to solving this problem more efficiently is to note that the configuration for a particular quantum (i.e., the fraction of I/O requests directed to a particular data distribution) is determined by the fraction complete, c. It is not affected by the choice of migration speed, M, during other quanta. Earlier quanta serve only to influence the time at which later ones begin, thus influencing how much they are discounted. Here, the exponential discounting is useful because it allows a plan to be shifted arbitrarily, by multiplying the discounted value by an appropriate factor. Delaying the start of a utility stream, $\mathrm{U}(t)$ by t_0 seconds:

$$\begin{aligned} U'_{PV} &= \int_{t_0}^{\infty} \mathrm{U}(t - t_0)\, \mathrm{e}^{-rt} dt \\ &= \int_{0}^{\infty} \mathrm{U}(s)\, \mathrm{e}^{-r(s+t_0)} ds \\ &= \mathrm{e}^{-rt_0} \int_{0}^{\infty} \mathrm{U}(s)\, \mathrm{e}^{-rs} ds \\ &= \mathrm{e}^{-rt_0} U_{PV} \end{aligned}$$

Since shifting the starting time is equivalent to multiplication by a constant factor, the optimal choice of migration speed for one quantum is not affected by an earlier quanta, because all possible choices are multiplied by that same non-negative factor.

While the optimal choice for one quantum is not affected by previous intervals, it is affected by subsequent ones. The choice of coordinator speed trades off the length of the current quantum (and the delay until the next one begins) against the utility value during the current quantum. The dependence on the future quanta but not on the past leads to an approach of calculating migration plans "in reverse." Starting with the final configuration (a 100% complete migration), the present value can be calculated as:

$$\begin{aligned} U_{PV,1} &= \int_{0}^{\infty} U_{final}\, \mathrm{e}^{-rt} dt \\ &= U_{final} \int_{0}^{\infty} \mathrm{e}^{-rt} dt \\ &= \frac{U_{final}}{r} \end{aligned}$$

The optimization is then defined recursively as:

$$U_{PV,c} = \max_{m:1\ldots M} \left(\mathrm{Q_d}\left(c,m\right) \cdot \mathrm{U}\left(c,m\right) + \mathrm{e}^{-r\mathrm{Q_d}(c,m)} U_{PV,c+q} \right)$$

This expression chooses the alternative with the highest present value based on a migration that begins at c percent complete. The first term is the duration of this quantum based on a particular multiprogramming level multiplied by the utility value for the quantum. The final term adds the (discounted) utility for the subsequent portions of the migration, shifting them by the duration of the current quantum. The output of this process is a present value for the data migration ($U_{PV} = U_{PV,0}$) and a list of multiprogramming values for each migration quantum. This process reduces the solution complexity to M/q or the number of multiprogramming levels times the number of migration quanta.

Migration of multiple datasets Building on the technique described above for migrating a single dataset, it is possible to use utility as a guide for controlling the migration of multiple datasets. For D datasets, there are $D!$ possible migration orderings, assuming a single dataset is moved at a time. The chosen approach is to proceed in a forward direction, migrating one dataset at a time, in a greedy fashion. Starting with the initial configuration, each of the possible datasets are evaluated, assuming that it is the only dataset that will be moved. This mimics a situation where exactly one dataset is moved and the migration process halted, leaving the system in that configuration. The single dataset migration that leads to the highest net present value is chosen as the first one to move. This new configuration then becomes the starting point for choosing the next dataset to move. This process is repeated until all datasets have been ordered. This requires $(D+1)D/2$ plans to be calculated, making the complexity $\mathrm{O}\left(D^2\right)$ instead of factorial. Unfortunately, this process of choosing an ordering is not guaranteed to produce an optimal plan. However, it chooses the largest gains first, capitalizing on benefits earlier rather than later. Additionally, if the administrator decides to halt a data migration while it is in progress, by biasing toward early gains, the system may have capitalized on the majority of the improvement.

Comparing alternative final configurations The procedures described above provide a method to generate a multi-dataset migration plan given a starting and ending configuration. Although their calculations account for the long-term utility of the final configuration, they do not provide a method

to generate one. Instead, they rely on the provisioning solver from previous chapters (constrained to use the existing hardware) to create suitable final configurations.

Allowing all datasets to be moved will produce the final configuration with the highest possible utility value, but every additional dataset that is moved increases the cost of the migration. Ideally, the search for a final configuration could be combined with the ordering and speed optimization. As a compromise, several attempts can be made to generate a good migration plan. By selecting only a subset of the datasets to move, the cost of migration can be decreased, and this choice of datasets could be influenced by recent storage events. For example, if a device fails, a good starting point would be to move only the datasets that had data fragments on that device. This selection could then be augmented with some number of (randomly chosen) additional datasets.

While storage systems are likely to be built using heterogeneous storage nodes, there will likely be only a few different types in a given system. Having large numbers of identical devices lead to having a number of equivalent system configurations. For example, in a system composed of two identical storage nodes and two datasets, if both datasets use a 1-of-1 declustered across 1 encoding, there are only two unique configurations (either they share a storage node or not). However, there are four total configurations. For initial provisioning, the existence of equivalent configurations matter very little because equivalent configurations produce identical utility. When planning migration, the final utility will be the same between these equivalent configurations, but the cost to migrate the data may differ. Because the current procedure for evaluating migration divides the choice of ending configuration and dataset ordering into separate steps, the cost of migration is not known to the solver that generates the final configuration. Returning to the example of 2 storage nodes and two datasets, if both datasets are initially stored on node one, and the final configuration is to have them utilize different nodes, depending on which of the equivalent final configurations are chosen, a different migration plan will be created (i.e., the choice of which dataset to move is determined by which of the equivalent configurations is chosen). One approach to handling this problem could be to include a re-organization step prior to calculating the migration plan. This step could swap storage nodes of the same type in an attempt to minimize the cost of migration. Approximations of this cost include the number of datasets or amount of data to move.

7.5 Migration summary

This chapter has described how utility can be used to guide the creation of plans for the online migration of data. It uses the Net Present Value of predicted utility to compare different migration

plans. This method allows plans that take varying amounts of time and result in different ending configurations to be compared, and it contains a configurable discount rate to adjust the weight placed on short-term versus long-term utility.

A method of generating migration plans with high utility is also presented. It relies on a series of nested optimizations that include the provisioning solver, a dataset ordering heuristic, and a method for determining the optimal migration speed for each dataset.

Using the techniques described here, different migration choices can be generated and compared. Additionally, by comparing them to doing nothing, it can be determined whether any of the optimization plans should be implemented.

Chapter 8

Automatic adaptation

The previous chapter described how utility can be used to evaluate data migration and re-encoding trade-offs. Those methods will be applied to several scenarios, showing how they can be used to guide automatic adaptation of the storage system. As a baseline, a discount rate of $1.65\text{x}10^{-6}$ s^{-1} will be used, corresponding to an expected once per week change in workloads or system components. The system will be permitted to optimize the number of outstanding requests made by the migration coordinator, but the value will be limited to between one and ten simultaneous requests to limit the computation requirements.

8.1 Effect of discount rate

The choice of discount rate affects how short-term utility is balanced with long-term utility. For example, with a large discount rate, migration choices will favor scenarios that increase current utility, potentially slowing migration operations. To illustrate this trade-off, several scenarios are compared, with each differing only in the discount rate that is applied. All scenarios will use the same workload and storage system configuration as well as the same utility function. The system will be similar to that of the 0.01¢ scenario in Section 6.1. There will be only a single dataset and corresponding workload, and there will be only seven storage nodes. The dataset begins in a 1-of-1 encoding on the first storage node and is re-encoded into the optimal, a 1-of-3 declustered across 7 encoding across all seven storage nodes. Figure 8.1 shows the utility throughout the migration process for four different discount rates, using 1% of the migration as the quanta size. Higher discount rates tend to slow the migration process because they place more weight on the current

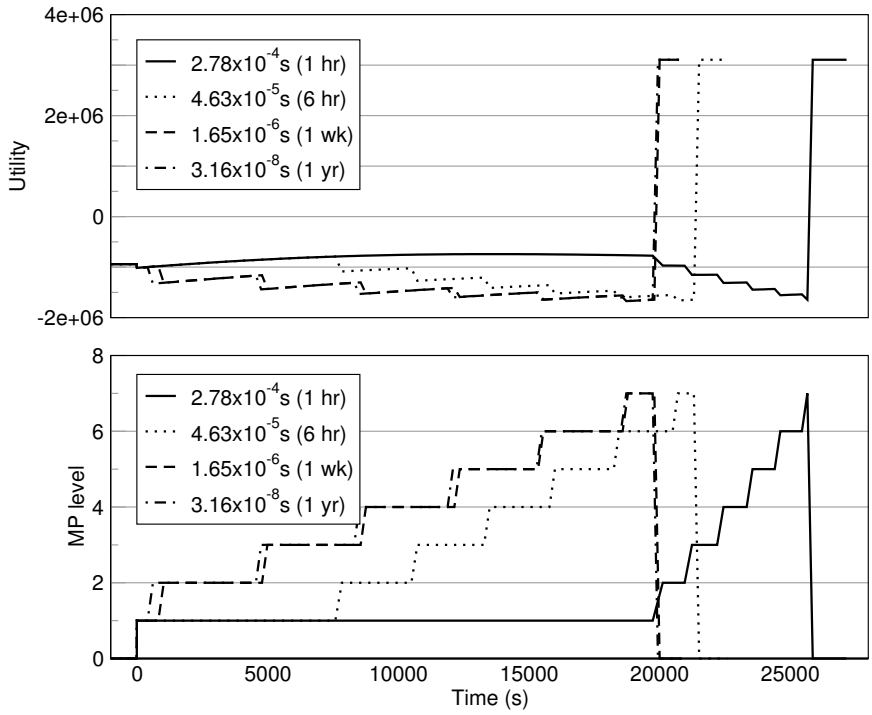


Figure 8.1: Migration using four different discount rates – This graph shows the predicted utility during migration using different discount rates to calculate the plan. Each of the discount rates are represented by the amount of time required to decay the value by 50% (i.e., 1 week corresponds to $r = 1.65\text{x}10^{-6}$). As the rate is lowered (a longer decay time), migrations tend to proceed quicker because the long-term utility is valued more. Once the time to decay by 50% significantly exceeds the duration of the data migration, it has little effect on the plan (e.g., 1 week and 1 yr have nearly identical plans). The large step upward at the end of the migration is largely a result of the increased availability and reliability since the old data distribution is no longer necessary for data access. The second graph shows the multiprogramming level of the coordinator during the migration. The higher this level, the more aggressively data is being moved.

utility. However, once the time period to decay by 50% (determined by the inverse of the discount rate) significantly exceeds the duration of the migration, it has little effect on the migration plan.

Also visible in this figure is the multiprogramming level of the coordinator. This value indicates the level of concurrency, or aggressiveness, of the migration coordinator. As the migration proceeds, this value tends to increase for several reasons. First, as the migration proceeds, the final configu-

ration (with its associated high utility) becomes discounted less, on a relative basis. This makes the optimization system more willing to sacrifice current utility. Second, as the migration progresses, the foreground contributes less to the movement of data, lengthening the time required to move a fixed amount of data at a given MP level, again changing the balance of time to completion versus current utility.

Beyond just affecting the rate of migration, the discount rate can determine whether a migration should be conducted at all. For example, consider the same scenario as above but beginning with a 1-of-3 declustered across 6 encoding. The initial utility is $3.09 M/yr, which is only 0.5% lower than the optimal configuration. For a low discount rate (e.g., the time to decay by 50% is one year), the migration cost can be offset by the small improvement provided by the new, final configuration. With higher discount rates (e.g., a decay time of one week), the end configuration is not sufficiently better to justify the migration cost.

8.2 Incremental provisioning

Once a storage system is provisioned and deployed, it is common for an organization to add additional datasets as the need arises. The system administrator is presented with a request to store one or more additional datasets, and he needs to add them to the existing storage infrastructure. When adding additional datasets to a system that is already in-service, the administrator is typically left with the dilemma of deciding whether to simply add the new datasets, fitting them around the existing ones or to migrate the existing data to hopefully obtain a more optimal final configuration. Utility allows the administrator to take a structured approach to this decision.

To illustrate the use of utility to guide this incremental provisioning task, this section will work through an example scenario. The initial configuration is identical to that of Section 6.1. Using the 0.01¢ per I/O utility function, it was shown that the two datasets should use a 1-of-3 declustered across 7 data encoding, where each of the two datasets reside on their own set of seven storage nodes. In this scenario, the administrator is presented with two additional datasets and workloads to be added to this storage system. For simplicity, these new dataset and workload specifications are identical to the original. The administrator must now decide how to arrange the four datasets on the storage system, given the restriction that there is a cost associated with moving the two existing ones.

It can be determined that the best placement for the new datasets *without* moving the existing ones is to use a 1-of-2 declustered across 3 encoding for the new datasets, with both interfering with

a single one of the original datasets. For example, if the original datasets used nodes 1–7 and 8–14, respectively, the new datasets could use 1–3 and 4–6, respectively. If the administrator chooses to move the existing datasets, the provisioning solver suggests that each use a 1-of-2 declustered across 3 encoding, with each dataset on its own set of disks. This leaves two storage nodes empty (i.e., it only uses $4 \cdot 3 = 12$ of the existing 14).[1] The configuration produced by moving the existing datasets has a higher utility (\$11.9 M/yr) than when the existing datasets remain in place (\$9.8 M/yr).[2] The dilemma for the administrator is whether this benefit is worth the cost of migrating data.

Assuming that the new datasets must be deployed immediately (before any reconfiguration), the initial deployment will use the distribution discussed above (with the two new datasets using 1/2/3 and interfering with a single existing workload). The analysis then compares the present value of keeping this configuration against the present value of migrating the data such that they all use a 1-of-2 declustered across 3 encoding. If the initial configuration is kept, the discounted value is:

$$\begin{aligned} U_{PV} =& \frac{U}{r} \\ =& \frac{\$9.8\text{x}10^{6}\ \text{yr}^{-1}}{1.65\text{x}10^{-6}\ \text{s}^{-1}} \left(\frac{\text{yr}}{365.25\ \text{day}}\right) \left(\frac{\text{day}}{24\ \text{hr}}\right) \left(\frac{\text{hr}}{3600\ \text{s}}\right) \\ =& \$1.89\text{x}10^{5} \end{aligned}$$

After calculating the migration plan for the original two datasets (the two new datasets do not need to be moved), the discounted value is \$3.29x10^5, which is greater than keeping the initial configuration, indicating that the migration should be performed.[3]

These same techniques that are used to decide if the existing data should be moved could be used to evaluate whether the new datasets should be deployed before or after the migration of the existing ones. This analysis would need to incorporate a penalty in the utility function related to the unavailability of the new data until migration is complete, but the methodology is the same.

Figure 8.2 shows the predicted utility and progression of the migration. The top graph shows the expected utility over the course of the migration. Also shown in this graph are the utility values for the initial and final configurations. These values can be seen as the small region to the left of time zero and the plateau that begins at $41,300$ s, respectively. The initial increase in utility as the

[1] Experiments with one dataset and four storage nodes confirm that 1/2/3 is preferred to */*/4 distributions.

[2] These utility values do not include the purchase cost of the storage nodes since the system has already been deployed.

[3] It is important to remember that the static analysis of a system configuration (e.g., \$9.8 M/yr) is only an instantaneous rate and assumes no discounting. As such, it cannot be directly compared to the result of a present value calculation.

migration begins, at $t = 0$, is the result of dataset 1 being served by a larger number of storage nodes (those serving the old data distribution and those serving the new). These extra resources cause a temporary increase in workload 1's performance. The second graph shows the throughput of each workload as the migration progresses. The third graph shows the rate at which blocks are being migrated. This rate is a combination of the writes from the foreground workload to blocks that still contain back-pointers and one half of the I/O rate of the coordinator. This graph slopes downward because the foreground workload contributes less to data movement as the migration progresses since fewer blocks are being written to the new data distribution for the first time. The fourth graph shows the multiprogramming level (MPL) of the migration coordinator. This is analogous to the maximum number of simultaneous requests that the coordinator may have outstanding. It is an indication of the speed with which the coordinator is attempting to migrate data. When the speed (multiprogramming level) of the coordinator increases, there is an associated increase in its I/Os per second (the fifth graph) and the number of blocks per second that are migrated (the third graph). This also negatively impacts the foreground workload because the coordinator is competing for resources (second graph). As a result of the decrease in foreground performance, overall (instantaneous) utility is also decreased (first graph).

Examining the behavior of the coordinator in the fourth graph, it can be seen that as each dataset is migrated, the coordinator becomes more aggressive. The MPL during each step of the migration is chosen to maximize the NPV of utility. Higher MPL will hurt current utility more but finish the migration more quickly. With an MPL of one, the first quantum of the first dataset will complete in 132 s. With an MPL of ten, this time can be cut to 62 s. This 53% decrease in time is accompanied by an approximately 20% decrease in utility as compared to an MPL of one. For the final quantum of this dataset's migration, this trade-off is considerably different. An MPL of one will complete the final quantum in 566 s, while an MPL of ten requires only 90 s. It causes nearly the same 20% decrease in utility, but it results in an 84% reduction in the migration time.

At both points in the migration process, the difference in IO/s of the coordinator between these two settings (MPL of one versus ten) is approximately six times. The ten times increase in parallelism resulting in an only six times increase in throughput is a result of the added latency and back-pressure on the closed-loop system. Comparing the overall IO/s of the coordinator for a given MPL between the start and end of the migration shows only a 10% throughput decrease, meaning that the ability for the coordinator to move data is largely independent of the fraction of the migration that has been completed.

The large differences in the utility/speed trade-off can be explained by the contribution of the

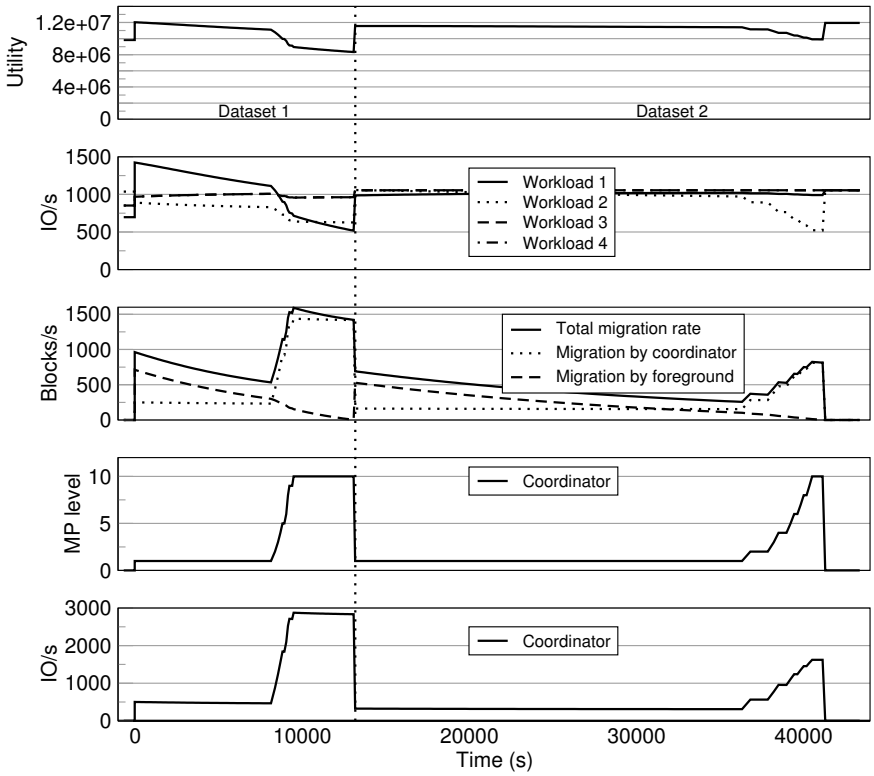


Figure 8.2: Migration plan for incremental provisioning – This set of graphs shows the migration of the two existing datasets (datasets one and two) for the incremental provisioning scenario. Dataset 1 is the first to be moved, followed by dataset 2. These two migrations are separated by the dotted line that runs through the graphs. The first graph shows utility as the migration proceeds. The second graph shows the I/O rates for each of the four workloads running in the system. The third graph shows the rate at which data is being migrated, in blocks per second. This rate is the sum of the blocks moved by the coordinator and the blocks moved by writes from the foreground workload. The downward slope to this graph is caused by the diminishing contribution from the foreground workload as the migration progresses. The fourth graph indicates the speed at which the coordinator is attempting to move data, given by its multiprogramming level (i.e., the maximum outstanding requests). The final graph shows the actual I/O rate of the coordinator. This I/O rate is evenly divided between reads of the old data distribution and writes to the new data distribution, making its corresponding contribution to the overall data migration (in blocks per second) one half of this value.

foreground workload. When the migration begins, all foreground writes plus all coordinator writes (one half of its IO/s) migrate a block. At the end of the migration, the foreground contributes almost nothing to the migration, relying completely on the coordinator. As the MPL is increased, it only affects (to a first approximation) the contribution of the coordinator, making a larger percentage difference when it is the only source of data migration. These two contributors to the data migration rate can be seen in the third graph of Figure 8.2. This graph shows the total migration rate (in blocks per second) and the individual contributions of the coordinator and foreground workload. The migration rate from the coordinator is exactly half of its IO/s (shown in the fifth graph), while the contribution of the foreground workload declines in direct proportion to the fraction of the migration that is complete. This foreground contribution is $(1-c) \cdot FracWrite \cdot IOPS_{FG}$. Page 95 provides the complete formula for calculating the migration rate.

To estimate the effect of choosing an incorrect coordinator MPL, the total accumulated utility during the migration (of both datasets, in this case) was compared against a migration in which the worst MPL was chosen during each step. This "worst-case" plan accumulated 6% less utility (non-discounted).

8.3 Repair

Many storage systems will suffer a hardware failure at some time during their service lifetime. This experiment shows how utility can prioritize recovery after a storage node failure.

In this scenario, the same 14-node storage system with two datasets is used. The utility functions and workloads are the same as the 0.01¢ scenario, described previously, except the utility function values for dataset/workload one's performance, availability, and reliability are doubled. This doubling increases the importance of dataset and workload number one over that of the other, but it attempts to maintain the relative importance of the different storage metrics within the workload by keeping the same ratio between them. The initial configuration for the datasets are to use a 1-of-3 declustered across 7 encoding on separate storage nodes. Beginning with this configuration, two storage nodes are assumed to fail simultaneously, one affecting each of the datasets. This effectively transforms the datasets into (approximately) 1-of-2 declustered across 6 data encodings.

Re-optimizing the configuration with the new, reduced number of storage nodes indicates that the best encoding is for both datasets to use a 1-of-3 declustered across 6 encoding. Figure 8.3 compares the two possible orderings for the repair operation. As expected, repairing dataset one before dataset two produces higher utility because it is "more important" than dataset two. This

ordering, however, has only 0.1% higher utility because the actual migration is so short (less than eight hours) relative to the discount rate which decays by half at one week. For migrating dataset one, then dataset two, the migration takes $2.6\text{x}10^5$ seconds, and the utility of the final configuration is \$9.28 M/yr. This means that the final configuration contributes $\$2.48\text{x}10^5$ to the present value of utility:

$$\begin{aligned} U_{PV,final} =& \frac{U_{final}}{r} \mathrm{e}^{-rT} \\ =& \frac{\$9.28\text{x}10^6 \text{ yr}^{-1}}{1.15\text{x}10^{-6} \text{ s}^{-1}} \mathrm{e}^{-1.15\text{x}10^{-6} \cdot 2.64\text{x}10^4} \left(\frac{\text{yr}}{365.25 \text{ day}}\right) \left(\frac{\text{day}}{24 \text{ hr}}\right) \left(\frac{\text{hr}}{3600 \text{ s}}\right) \\ =& \$2.48\text{x}10^5 \end{aligned}$$

The overall utility of this configuration is $\$2.53\text{x}10^5$. Therefore, the final configuration accounts for 98% of the present value of utility. While this result suggests that migrations of such short duration are relatively insensitive to the dataset ordering, it is important to note that in the example presented here, the datasets were segregated, making the individual migration steps insensitive to the ordering. It is possible to construct scenarios where the ordering of migration influences not only the utility but also the duration of migration because of contention for storage nodes.

8.4 Time to upgrade?

Over time, the demands placed on a storage system change. A previous example demonstrated how utility can be used to handle incremental provisioning, helping an administrator decide how to incorporate new datasets and workloads into an existing system. Another common problem for system administrators is that existing workloads tend to change over time as user populations change and applications become more or less popular. When these changes occur, the administrator must decide how the storage system's configuration should be modified to serve these new demands. He must decide whether a simple re-organization of data will suffice or if it is time to upgrade the system's hardware.

This scenario demonstrates how utility can be used to aid the administrator in adjusting a storage system to workload changes. Beginning with the configured storage system from Section 6.1 and the 0.01¢ utility function, it has been shown that the 1-of-3 declustered across 7 encoding is optimal for the two datasets. However, if one of the workloads increases in intensity by doubling the number of outstanding requests, decreasing its inter-request think time, and slightly increasing its sequentiality

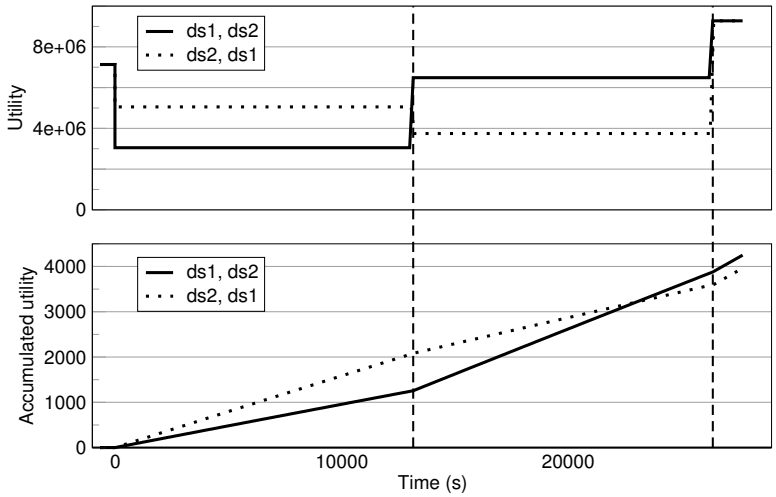


Figure 8.3: Comparison of two possible orderings for repair – Both orderings use the same initial and final data distributions, and they take approximately the same amount of time to complete. The utility difference between them is based solely on the migration order. By migrating the higher value dataset first (dataset one), the gain in utility is realized sooner. The bottom graph shows the "accumulation" of utility as the migration proceeds (the integral of the top graph). This bottom graph shows that once the higher value dataset is migrated, the system begins accumulating utility at a higher rate, overtaking the other migration ordering. Since the end configuration is identical for both orderings, the gap between the two will remain constant once the migration is complete.

as well as its ratio of reads, how should the administrator respond?[4] He has the option of doing nothing, re-encoding one or both of the datasets, or he may even purchase additional storage nodes to compensate for the additional load. The current configuration (with the more intense workload) has a utility of $\$1.01\text{x}10^7$/yr which produces a present value of $\$2.80\text{x}10^5$. Optimizing the system with the current hardware yields a configuration in which the dataset for the more intense workload uses a 1-of-3 declustered across 8 encoding while the less accessed dataset is changed to a 1-of-3 declustered across 6 encoding. This optimization shifts a storage node to the more intense workload. After accounting for the migration, this option has a present value of $\$2.82\text{x}10^5$. When additional hardware can be purchased, the solver produces a configuration that uses one additional storage node, keeping the dataset from the unaltered workload in its original configuration while changing

[4]See the "rrw2" workload in Table A.3 for the full specification of this new workload.

the more intense workload's dataset to a 1-of-3 declustered across 8 encoding. This option also produces a present value of $\$2.82x10^5$.

Strictly interpreting the results, both of the new configurations are better than maintaining the current. However, expressing objectives (i.e., creating utility functions) and creating system models are subject to some amount of uncertainty. Based on this uncertainty, spending resources either to upgrade the system or even to optimize the existing configuration may not have a discernible benefit. Exploring the impact of uncertainty on these optimization decisions is an interesting area for future research.

Chapter 9

Conclusions

This dissertation set out to demonstrate that "utility functions are a promising method for guiding the design and optimization of a distributed, self-tuning storage system." In support of that goal, this document shows the importance of making trade-offs across storage metrics, and it explains how utility can be used to guide those trade-offs, during both initial provisioning and online adaptation. Further, it demonstrates prototype optimization systems that are guided by utility and shows how they can be used to optimize several scenarios.

9.1 Contributions

This work provides evidence that automatic provisioning and tuning systems need to be able to make trade-offs across storage system metrics. Since most configuration choices affect multiple system metrics, this flexibility allows an automated system to produce superior solutions because it can better balance costs and benefits.

Chapters 1–4 develop a framework for using utility to evaluate storage system configurations. This tuning loop provides a plug-in architecture that allows more detailed models and additional storage metrics to be added easily. The chapters also present the models that were used for analysis, showing the robustness of the architecture by including both very simple models that were easy to create as well as more detailed models such as the queueing-based performance model.

Chapters 5 and 6 show how utility can be used to guide storage provisioning. They develop a solver that is able to create good storage solutions based on the guidance provided by utility functions, showing that utility can be used successfully. The chapters also present several scenarios

to highlight the importance of using utility for provisioning and the benefits that it has over more traditional, automated approaches.

Chapters 7 and 8 build on the provisioning tool to show how utility can be used to guide the optimization of a storage system once it has been deployed. They present a methodology for creating optimization plans and evaluating them against each other as well as comparing them to maintaining the current configuration. This methodology provides a framework that allows a system to automatically decide from among several competing plans and to determine whether any of them should be acted upon. These chapters also present a method for creating optimization plans, controlling the speed and order of data migration based on guidance provided by a utility function. This framework was applied to several scenarios to show how it could be used for several common storage optimization tasks.

9.2 Important future work

While this work shows the potential for utility-guided provisioning and optimization, it has highlighted several interesting areas of future work. One of the initial assumptions of this work is that an administrator (or their agent) can provide a utility function for use by the system. While this work presents an overview of how monetary costs and benefits can be used to create this function, it is not a simple task. There are consequences that are difficult to quantify, such as user satisfaction or impact on the organization's reputation. The accuracy of the end solution depends, in part, on the accuracy and completeness of the utility function. Pursuing techniques for creating good utility functions is important. In addition to more detailed financial and actuarial analysis, research on elicitation techniques [Boutilier et al., 2003; Braziunas and Boutilier, 2005; Chen and Pu, 2004; Patrascu et al., 2005] also shows promise.

The construction of the solvers used in this work rely on having good storage models that can analyze a potential configuration and produce accurate storage metrics. There is, typically, a tension between the accuracy and speed of system models. Unfortunately, both are a priority. While some work, such as that by Anderson [2001], has targeted both, more work is needed across many of the storage metrics.

Another area for improvement in system modeling is the ability to characterize workloads. Specifically, being able to tell when a workload has "changed" is very valuable. Changes to both hardware and system objectives are relatively easy to notice because they are (usually) explicit

events, and they tend to be stable. There does not appear to be an obvious indicator for workload changes nor an easy way to quickly filter temporary fluctuations.

As automated approaches to system configuration and tuning become more common, it is important to create methods for layering these systems. For example, this work has concentrated on configuring storage systems by making trade-offs across storage metrics. The administrator would like to express objectives not as storage metrics but as application-level metrics. Additionally, multiple layers of applications could be involved in providing the end functionality to users, and it is important to develop techniques that can permit trade-offs across these different levels.

Appendix A

Description of modeled components

The following tables list the attributes for each of the components (clients, storage nodes, and workloads) that were used in the evaluation. These attributes define the low-level characteristics of the components and are used by the system models to predict the values of high-level storage system metrics.

Table A.1: Clients used in experiments – This table lists the attributes of the clients that were used in the evaluations and case studies. The client is based on measurements from a 2.66 GHz Pentium 4.

	Client
Name	standard
CPU encoding time	0.2 ms
Network latency	125 μs
Network bandwidth	119 MB/s

Table A.2: Storage nodes used in experiments – This table lists the attributes of the storage nodes that were used in the evaluations and case studies.

	Storage nodes			
Name	15k73	s500	s500c	s500e
AFR	0.0062	0.015	0.015	0.015
Availability	0.98	0.95	0.95	0.95
Capacity	73.4 GB	500 GB	500 GB	500 GB
Disk bandwidth	100 MB/s	70 MB/s	70 MB/s	70 MB/s
Disk latency	5.75 ms	5.5 ms	5.5 ms	5.5 ms
Network bandwidth	119 MB/s	119 MB/s	119 MB/s	119 MB/s
Network latency	125 μs	125 μs	125 μs	125 μs
Power	50 W	50 W	50 W	50 W
Purchase cost	$300	$5000	$2,000	$10,000

Table A.3: Workloads used in experiments – This table lists the attributes of the workloads that were used in the evaluations and case studies. The "6450" workload is an arbitrary 50/50 read/write mix. The "rrw" workloads are small, random I/O such as from an OLTP-type workload. The "sread" workload is a sequential read workload, as would be expected from I/O trace processing.

	Workloads			
Name	6450	rrw	rrw2	sread
I/O size	64 kB	8 kB	8 kB	32 kB
Multiprogramming level	32	5	10	10
Random fraction	0.5	0.95	0.8	0.0
Read fraction	0.5	0.5	0.7	1.0
Think time	50 ms	1 ms	0.8 ms	1 ms

Bibliography

Numbers at the end of an entry refer to the pages on which that entry is cited.

Abd-El-Malek, M., Courtright II, W. V., Cranor, C., Ganger, G. R., Hendricks, J., Klosterman, A. J., Mesnier, M., Prasad, M., Salmon, B., Sambasivan, R. R., Sinnamohideen, S., Strunk, J. D., Thereska, E., Wachs, M., and Wylie, J. J. (2005). Ursa Minor: Versatile cluster-based storage. In *Conference on File and Storage Technologies (FAST)*, pages 59–72. USENIX Association. 4, 5, 15, 19

Alvarez, G. A., Uysal, M., and Merchant, A. (2001a). Efficient verification of performability guarantees. In *International Workshop on Performability Modeling of Computer and Communications Systems (PMCCS)*, pages 95–99. 39

Alvarez, G. A., Wilkes, J., Borowsky, E., Go, S., Romer, T. H., Becker-Szendy, R., Golding, R., Merchant, A., Spasojevic, M., and Veitch, A. (2001b). Minerva: An automated resource provisioning tool for large-scale storage systems. *ACM Transactions on Computer Systems*, 19(4):483–518. 2, 13

Amiri, K. and Wilkes, J. (1996). Automatic design of storage systems to meet availability requirements. Technical Report HPL–SSP–96–17, Hewlett-Packard Laboratories. 42

Anderson, E. (2001). Simple table-based modeling of storage devices. Technical Report HPL–SSP–2001–4, Hewlett-Packard Laboratories. 39, 112

Anderson, E., Hall, J., Hartline, J., Hobbs, M., Karlin, A. R., Saia, J., Swaminathan, R., and Wilkes, J. (2001a). An experimental study of data migration algorithms. In *International Workshop on Algorithm Engineering (WAE)*, pages 145–158. Springer-Verlag. 96

Anderson, E., Hobbs, M., Keeton, K., Spence, S., Uysal, M., and Veitch, A. (2002). Hippodrome: Running circles around storage administration. In *Conference on File and Storage Technologies (FAST)*, pages 175–188. USENIX Association. 2, 13

Anderson, E., Kallahalla, M., Keeton, K., Swaminathan, R., and Uysal, M. (2003). Application-centric integrated storage management. In *Algorithms and Architectures for Self-Managing Systems*, pages 19–20. ACM Press. 2, 11

Anderson, E., Kallahalla, M., Spence, S., Swaminathan, R., and Wang, Q. (2001b). Ergastulum: Quickly finding near-optimal storage system designs. Technical Report HPL–SSP–2001–05, Hewlett-Packard Laboratories. 13

Anderson, E., Spence, S., Swaminathan, R., Kallahalla, M., and Wang, Q. (2005). Quickly finding near-optimal storage designs. *ACM Transactions on Computer Systems*, 23(4):337–374. 2, 13, 79

Anderson, T. E., Dahlin, M. D., Neefe, J. M., Patterson, D. A., Roselli, D. S., and Wang, R. Y. (1996). Serverless network file systems. *ACM Transactions on Computer Systems*, 14(1):41–79. 15

Apthorpe, R. (2001). A probabilistic approach to estimating computer system reliability. In *Systems Administration Conference (LISA)*, pages 31–45. USENIX Association. 42

Asami, S. (2000). Reducing the cost of system administration of a disk storage system built from commodity components. PhD thesis UCB/CSD–00–1100, EECS Department, University of California, Berkeley. 26

AuYoung, A., Grit, L., Wiener, J., and Wilkes, J. (2006). Service contracts and aggregate utility functions. In *International Symposium on High-Performance Distributed Computing (HPDC)*, pages 119–131. IEEE. 14, 50

Bhagwan, R., Douglis, F., Hildrum, K., Kephart, J. O., and Walsh, W. E. (2005). Time-varying management of data storage. In *Workshop on Hot Topics in System Dependability (HotDep)*. IEEE. 51

Borowsky, E., Golding, R., Merchant, A., Schreier, L., Shriver, E., Spasojevic, M., and Wilkes, J. (1997). Using attribute-managed storage to achieve QoS. In *International Workshop on Quality of Service (IWQoS)*, pages 203–207. IFIP. 2, 11

Boutilier, C., Das, R., Kephart, J. O., Tesauro, G., and Walsh, W. E. (2003). Cooperative negotiation in autonomic systems using incremental utility elicitation. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 89–97. Association for Uncertainty in Artificial Intelligence. 112

Braziunas, D. and Boutilier, C. (2005). Local utility elicitation in GAI models. In *Conference on Uncertainty in Artificial Intelligence (UAI)*. Association for Uncertainty in Artificial Intelligence. 112

Brindle, A. (1981). Genetic algorithms for function optimization. PhD thesis TR81–2, Department of Computing Science, University of Alberta. 68

Brown, A. and Patterson, D. A. (2000). Towards availability benchmarks: A case study of software RAID systems. In *USENIX Annual Technical Conference*. USENIX Association. 26

Browning, E. K. and Zupan, M. A. (1996). *Microeconomic Theory and Applications*. HarperCollins, fifth edition. 6

Burkhard, W. A. and Menon, J. (1993). Disk array storage system reliability. In *Symposium on Fault-Tolerant Computer Systems (FTCS)*, pages 432–441. IEEE. 42

Candea, G. and Fox, A. (2002). A utility-centered approach to building dependable infrastructure services. In *ACM SIGOPS European Workshop*. ACM Press. 50

Chen, L. and Pu, P. (2004). Survey of preference elicitation methods. Technical Report IC/200467, Swiss Federal Institute of Technology in Lausanne (EPFL). 59, 112

Chen, Y., Das, A., Gautam, N., Wang, Q., and Sivasubramaniam, A. (2004). Pricing and autonomic control of web servers with time-varying request patterns. In *International Conference on Autonomic Computing (ICAC)*, pages 290–291. IEEE. 50

Chu, P. C. and Beasley, J. E. (1997). A genetic algorithm for the generalised assignment problem. *Computers & Operations Research*, 24(1):17–23. 68

Contingency Planning Research (1996). The cost of downtime. http://www.contingencyplanningresearch.com/codgraph.pdf. Accessed April, 2008. 10

Corcoran, A. L. and Hale, J. (1994). A genetic algorithm for fragment allocation in a distributed database system. In *Symposium on Applied Computing (SAC)*, pages 247–250. ACM Press. 66

Douceur, J. R. and Wattenhofer, R. P. (2001a). Large-scale simulation of replica placement algorithms for a serverless distributed file system. In *International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS)*, pages 311–319. IEEE. 26

Douceur, J. R. and Wattenhofer, R. P. (2001b). Optimizing file availability in a secure serverless distributed file system. In *Symposium on Reliable Distributed Systems (SRDS)*. IEEE. 26

Eagle Rock Alliance, Ltd. (2001). Online survey results: 2001 cost of downtime. http://contingencyplanningresearch.com/2001~Survey.pdf. Accessed April, 2008. 10

Eisler, M., Corbett, P., Kazar, M., Nydick, D. S., and Wagner, J. C. (2007). Data ONTAP GX: A scalable storage cluster. In *Conference on File and Storage Technologies (FAST)*, pages 139–152. USENIX Association. 15

Energy Information Administration (2007). Average retail price of electricity to ultimate customers by end-use sector, by state. http://www.eia.doe.gov/cneaf/electricity/epm/table5_6_a.html. 53

Feitelson, D. G. and Naaman, M. (1999). Self-tuning systems. *IEEE Software*, 16(2):52–60. 66

Feltl, H. and Raidl, G. R. (2004). An improved hybrid genetic algorithm for the generalized assignment problem. In *Symposium on Applied Computing (SAC)*, pages 990–995. ACM Press. 68

Frølund, S. and Koistinen, J. (1998). Quality of service specification in distributed object systems design. In *Conference on Object-Oriented Technologies and Systems (COOTS)*, pages 1–18. USENIX Association. 11

Ganger, G. R., Strunk, J. D., and Klosterman, A. J. (2003). Self-* storage: Brick-based storage with automated administration. Technical Report CMU–CS–03–178, Carnegie Mellon University. 15

Gaonkar, S., Keeton, K., Merchant, A., and Sanders, W. H. (2006). Designing dependable storage solutions for shared application environments. In *International Conference on Dependable Systems and Networks (DSN)*, pages 371–382. IEEE Computer Society. 12, 14, 51

Gartner Consulting (2000). Total cost of storage ownership — a user-oriented approach. Research note, Gartner Group, Inc. 10

Geist, R. and Trivedi, K. (1993). An analytic treatment of the reliability and performance of mirrored disk subsystems. In *Symposium on Fault-Tolerant Computer Systems (FTCS)*, pages 442–450. IEEE. 39

Gelb, J. P. (1989). System-managed storage. *IBM Systems Journal*, 28(1):77–103. 2, 11

Ghemawat, S., Gobioff, H., and Leung, S.-T. (2003). The Google file system. In *Symposium on Operating System Principles (SOSP)*, pages 29–43. ACM Press. 15

Ghosh, S., Hansen, J., and Rajkumar, R. (2003). Scalable resource allocation for multi-processor QoS optimization. In *International Conference on Distributed Computing Systems (ICDCS)*, pages 174–183. IEEE. 50

Gibson, G. A. (1991). Redundant disk arrays: Reliable, parallel secondary storage. PhD thesis UCB/CSD–91–613, EECS Department, University of California, Berkeley. 42

Gibson, G. A., Nagle, D. F., Amiri, K., Butler, J., Chang, F. W., Gobioff, H., Hardin, C., Riedel, E., Rochberg, D., and Zelenka, J. (1998). A cost-effective, high-bandwidth storage architecture. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 92–103. ACM Press. 15

Goodson, G. R., Wylie, J. J., Ganger, G. R., and Reiter, M. K. (2003). A protocol family for versatile survivable storage infrastructures. Technical Report CMU–PDL–03–103, Parallel Data Lab, Carnegie Mellon University. 17

Goodson, G. R., Wylie, J. J., Ganger, G. R., and Reiter, M. K. (2004). Efficient Byzantine-tolerant erasure-coded storage. In *International Conference on Dependable Systems and Networks (DSN)*, pages 135–144. IEEE Computer Society. 16

Gray, J. (2003). A conversation with Jim Gray. *ACM Queue*, 1(4):8–17. 10

Gunther, N. and Harding, P. (2007). PDQ (pretty damn quick) version 4.2. `http://www.perfdynamics.com/Tools/PDQ.html`. Accessed April, 2008. 32, 38

Gunther, N. J. (2005). *Analyzing Computer System Performance with Perl::PDQ*. Springer-Verlag. 32, 37

Howard, J. H., Kazar, M. L., Menees, S. G., Nichols, D. A., Satyanarayanan, M., Sidebotham, R. N., and West, M. J. (1988). Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81. 15

Irwin, D. E., Grit, L. E., and Chase, J. S. (2004). Balancing risk and reward in a market-based task service. In *International Symposium on High-Performance Distributed Computing (HPDC)*, pages 160–169. IEEE. 14, 50, 54

Karlsson, M., Karamanolis, C., and Zhu, X. (2004). Triage: Performance isolation and differentiation for storage systems. In *International Workshop on Quality of Service (IWQoS)*, pages 67–74. IFIP. 88

Keeney, R. L. and Raiffa, H. (1993). *Decisions with Multiple Objectives: Preferences and Value Tradeoffs*. Cambridge University Press. 6, 48

Keeton, K., Beyer, D., Brau, E., Merchant, A., Santos, C., and Zhang, A. (2006). On the road to recovery: Restoring data after disasters. In *European Systems Conference (EuroSys)*, pages 235–248. ACM Press. 12, 13, 14, 51

Keeton, K. and Merchant, A. (2004). A framework for evaluating storage system dependability. In *International Conference on Dependable Systems and Networks (DSN)*, pages 877–886. IEEE Computer Society. 13

Keeton, K., Santos, C., Beyer, D., Chase, J., and Wilkes, J. (2004). Designing for disasters. In *Conference on File and Storage Technologies (FAST)*, pages 59–72. USENIX Association. 12, 13, 14, 51

Keeton, K. and Wilkes, J. (2002). Automating data dependability. In *ACM SIGOPS European Workshop*, pages 93–100. ACM Press. 13, 51

Kelly, T. (2003). Utility-directed allocation. In *Algorithms and Architectures for Self-Managing Systems*, pages 47–52. ACM Press. 50

Kephart, J. O. and Das, R. (2007). Achieving self-management via utility functions. *IEEE Internet Computing*, 11(1):40–48. 50

Kephart, J. O. and Walsh, W. E. (2004). An artificial intelligence perspective on autonomic computing policies. In *International Workshop on Policies for Distributed Systems and Networks (POLICY)*, pages 3–12. IEEE. 2, 50

Lee, C., Lehoczky, J., Siewiorek, D., Rajkumar, R., and Hansen, J. (1999). A scalable solution to the multi-resource QoS problem. In *Real-Time Systems Symposium (RTSS)*, pages 315–326. IEEE. 50

Lee, E. K. and Thekkath, C. A. (1996). Petal: Distributed virtual disks. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 84–92. ACM Press. 15

Lee, J. Y. B. and Leung, R. W. T. (2002). Design and analysis of a fault-tolerant mechanism for a server-less video-on-demand system. In *International Conference on Parallel and Distributed Systems (ICPADS)*, pages 489–494. IEEE Computer Society. 42

Litzkow, M., Livny, M., and Mutka, M. (1988). Condor - a hunter of idle workstations. In *International Conference on Distributed Computing Systems (ICDCS)*, pages 104–111. IEEE. iii

Loaiza, J. (2000). Optimal storage configuration made easy. Whitepaper 295, Oracle Corporation. 12

Lu, C., Alvarez, G. A., and Wilkes, J. (2002). Aqueduct: Online data migration with performance guarantees. In *Conference on File and Storage Technologies (FAST)*, pages 219–230. USENIX Association. 88

Mesnier, M. P., Wachs, M., Sambasivan, R. R., Zheng, A., and Ganger, G. R. (2007). Modeling the relative fitness of storage. In *Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 37–48. ACM Press. 39

Meyer, J. F. (1980). On evaluating the performability of degradable computing. *IEEE Transactions on Computers*, C-29(8):720–731. 39

Mitchell, T. M. (1997). *Machine Learning*. McGraw-Hill. 65

Nagle, D., Serenyi, D., and Matthews, A. (2004). The Panasas ActiveScale Storage Cluster: Delivering scalable high bandwidth storage. In *ACM/IEEE Conference on Supercomputing (SC)*, page 53. IEEE Computer Society. 64

Nicola, V. F. and Goyal, A. (1990). Modeling of correlated failures and community error recovery in multiversion software. *IEEE Transactions on Software Engineering*, 16(3):350–359. 25

Parekh, S., Gandhi, N., Hellerstein, J., Tilbury, D., Jayram, T. S., and Bigus, J. (2002). Using control theory to achieve service level objectives in performance management. *Real Time Systems Journal*, 23(1–2):127–141. 88

Pâris, J.-F., Schwarz, T. J. E., and Long, D. D. E. (2006). Evaluating the reliability of storage systems. Technical Report UH–CS–06–08, Department of Computer Science, University of Houston. 40

Patrascu, R., Boutilier, C., Das, R., Kephart, J. O., Tesauro, G., and Walsh, W. E. (2005). New approaches to optimization and utility elicitation in autonomic computing. In *National Conference on Artificial Intelligence (AAAI)*, pages 140–145. AAAI Press. 59, 112

Patterson, D. A. (2002). A simple way to estimate the cost of downtime. In *Systems Administration Conference (LISA)*, pages 185–188. USENIX Association. 55

Pinheiro, E., Weber, W.-D., and Barroso, L. A. (2007). Failure trends in a large disk drive population. In *Conference on File and Storage Technologies (FAST)*, pages 17–28. USENIX Association. 25

Saito, Y., Frølund, S., Veitch, A., Merchant, A., and Spence, S. (2004). FAB: building distributed enterprise disk arrays from commodity components. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 48–58. ACM Press. 15

Savage, S. and Wilkes, J. (1996). Afraid—a frequently redundant array of independent disks. In *Winter USENIX Technical Conference*, pages 27–39. USENIX Association. 26

Schroeder, B. and Gibson, G. A. (2007). Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you? In *Conference on File and Storage Technologies (FAST)*, pages 1–16. USENIX Association. 25

Shriver, E. (1999). A formalization of the attribute mapping problem. Technical Report HPL–1999–127, Hewlett-Packard Laboratories. 11

Siewiorek, D. P. and Swarz, R. S. (1982). *The Theory and Practice of Reliable System Design*. Digital Press. 22

Soules, C. A. N., Goodson, G. R., Strunk, J. D., and Ganger, G. R. (2003). Metadata efficiency in versioning file systems. In *Conference on File and Storage Technologies (FAST)*, pages 43–58. USENIX Association. 18

St.Pierre, E. (2007). ILM: Tiered services & the need for classification. Technical tutorial, Storage Networking Industry Association. http://www.snia.org/education/tutorials/2007/spring/data-management/ILM-Tiered_Services.pdf. 12

Strunk, C. W. (2003). *Measurement of Linear Nanometric Distances Using Scattered Evanescent Radiation*. PhD thesis, Department of Chemical Engineering, Carnegie Mellon University. ii

Strunk, J. D., Goodson, G. R., Scheinholtz, M. L., Soules, C. A. N., and Ganger, G. R. (2000). Self-Securing Storage: Protecting data in compromised systems. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 165–180. USENIX Association. 18

Telford, R., Horman, R., Lightstone, S., Markov, N., O'Connell, S., and Lohman, G. (2003). Usability and design considerations for an autonomic relational database management system. *IBM Systems Journal*, 42(4):568–581. 2

Tesauro, G. (2005). Online resource allocation using decompositional reinforcement learning. Technical Report RC23690, IBM Research. 50

Thereska, E., Abd-El-Malek, M., Wylie, J. J., Narayanan, D., and Ganger, G. R. (2006). Informed data distribution selection in a self-predicting storage system. In *International Conference on Autonomic Computing (ICAC)*, pages 187–198. IEEE. 32

Uttamchandani, S., Voruganti, K., Srinivasan, S., Palmer, J., and Pease, D. (2004). Polus: Growing storage QoS management beyond a "four-year old kid". In *Conference on File and Storage Technologies (FAST)*, pages 31–44. USENIX Association. 13

Uttamchandani, S., Yin, L., Alvarez, G. A., Palmer, J., and Agha, G. (2005). CHAMELEON: A self-evolving, fully-adaptive resource arbitrator for storage systems. In *USENIX Annual Technical Conference*, pages 75–88. USENIX Association. 88

Uysal, M., Alvarez, G. A., and Merchant, A. (2001). A modular, analytical throughput model for modern disk arrays. In *International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS)*, pages 183–192. 39

Varki, E., Merchant, A., Xu, J., and Qiu, X. (2004). Issues and challenges in the performance analysis of real disk arrays. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):559–574. 39

Walsh, W. E., Tesauro, G., Kephart, J. O., and Das, R. (2004). Utility functions in autonomic systems. In *International Conference on Autonomic Computing (ICAC)*, pages 70–77. IEEE. 50

Ward, J., O'Sullivan, M., Shahoumian, T., and Wilkes, J. (2002). Appia: automatic storage area network fabric design. In *Conference on File and Storage Technologies (FAST)*, pages 203–217. USENIX Association. 29

Weddle, C., Oldham, M., Qian, J., Wang, A.-I. A., Reiher, P., and Kuenning, G. (2007). PARAID: A gear-shifting power-aware RAID. *ACM Transactions on Storage*, 3(3):13. 28

Weikum, G., Zabback, P., and Scheuermann, P. (1990). Dynamic file allocation in disk arrays. Technical Report 147, Department of Computer Science, ETH Zurich. 12

Wilkes, J. (2001). Traveling to Rome: QoS specifications for automated storage system management. In *International Workshop on Quality of Service (IWQoS)*, pages 75–91. IFIP. 2, 11

Wilkes, J., Golding, R., Staelin, C., and Sullivan, T. (1996). The HP AutoRAID hierarchical storage system. *ACM Transactions on Computer Systems*, 14(1):108–136. 12

Wolf, J. (1989). The Placement Optimization Program: A practical solution to the disk file assignment problem. *Performance Evaluation Review*, 17(1):1–9. 12

Wong, T. M., Golding, R. A., Glider, J. S., Borowsky, E., Becker-Szendy, R. A., Fleiner, C., Kenchammana-Hosekote, D. R., and Zaki, O. A. (2005). Kybos: Self-management for distributed brick-based storage. Research Report RJ 10356, IBM Almaden Research Center. 15

Zhu, Q., Chen, Z., Tan, L., Zhou, Y., Keeton, K., and Wilkes, J. (2005). Hibernator: Helping disk arrays sleep through the winter. In *Symposium on Operating System Principles (SOSP)*, pages 177–190. ACM Press. 28