Using Utility to Provision Storage Systems

John D. Strunk[†], Eno Thereska[∗], Christos Faloutsos[†], Gregory R. Ganger[†]
[†]*Carnegie Mellon University*
[∗]*Microsoft Research, Cambridge UK*

Abstract

Provisioning a storage system requires balancing the costs of the solution with the benefits that the solution will provide. Previous provisioning approaches have started with a fixed set of requirements and the goal of automatically finding minimum cost solutions to meet them. Such approaches neglect the cost-benefit analysis of the purchasing decision.

Purchasing a storage system involves an extensive set of trade-offs between metrics such as purchase cost, performance, reliability, availability, power, etc. Increases in one metric have consequences for others, and failing to account for these trade-offs can lead to a poor return on the storage investment. Using a collection of storage acquisition and provisioning scenarios, we show that utility functions enable this cost-benefit structure to be conveyed to an automated provisioning tool, enabling the tool to make appropriate trade-offs between different system metrics including performance, data protection, and purchase cost.

1 Introduction

Whether buying a new car or deciding what to eat for lunch, nearly every decision involves trade-offs. Purchasing and configuring a storage system is no different. IT departments want solutions that meet their storage needs in a cost-effective manner. Currently, system administrators must rely on human "expertise" regarding the type and quantity of hardware to purchase as well as how it should be configured. These recommendations often take the form of standard configurations or "rules of thumb" that can easily lead to an expensive, over-provisioned system or one that fails to meet the customer's expectations. Because every organization and installation is unique, a configuration that works well for one customer may provide inadequate service or be too expensive for another. Proper storage solutions account not only for the needs of the customer's applications but also the customer's budget and cost structure.

Mechanisms	Goals	Utility
RAID-5 64 kB stripe size	500 IO/s 5 "nines"	U(*revenue, costs*)
Manual configuration	*Provisioning using fixed requirements*	*Provisioning using business objectives*

Expressiveness →

Figure 1: Utility provides value beyond mechanism-based and goal-based specification – 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 the service. This allows the systems to better match the cost and benefit structure of an organization.

Previous approaches to provisioning [5–7] have worked to create minimum cost solutions that meet some predefined requirements. Purchasing the cheapest storage system that meets a set of fixed requirements neglects the potential trade-offs available to the customer, because it separates the analysis of the benefits from that of the costs. In this scenario, system administrators are forced to determine their storage requirements (based on their anticipated benefits) prior to the costs becoming apparent — those "5 nines" are not free. Our goal is to combine these two, currently separate, analyses to produce more cost-effective storage solutions.

While some requirements may be non-negotiable, most are flexible based on the costs required to implement them. Few organizations would say, "I need 5000 IO/s, and I don't care what it costs." Performance objectives are related to employees' productivity and potential revenue. Data protection objectives are related to the cost, inconvenience, and publicity that come from downtime and repair. These underlying costs and benefits determine the ROI an organization can achieve, and a provisioning tool needs to consider both.

We propose using *utility functions*, instead of fixed requirements, as a way for the system administrator to

communicate these underlying costs and benefits to an automated provisioning tool. Figure 1 shows a spectrum of storage configuration methods from setting low-level mechanisms to communicating costs and benefits using utility. Utility functions provide a way to specify storage requirements in terms of the organization's cost/benefit structure, allowing an automated tool to provide the level of service that produces the most cost-effective storage solution for that environment. A utility function expresses the desirability of a given configuration — the benefits of the provided service, less the associated costs. The objective of the tool then becomes maximizing utility in the same way previous work has sought to minimize system cost.

This paper describes utility, utility functions, and the design of a utility-based provisioning tool, highlighting the main components that allow it to take high-level objectives from the administrator and produce cost-effective storage solutions. It describes the implementation and provides a brief evaluation of a prototype utility-based provisioning tool. The purpose of this tool is to show the feasibility of using utility to guide storage system provisioning. After this provisioning tool is shown to efficiently find good storage solutions, three hypothetical case studies are used to highlight several important benefits of using utility for storage provisioning. First, when both costs and benefits are considered, the optimal configuration may defy the traditional intuition about the relative importance of system metrics (e.g., performance vs. data protection). Second, utility can be used to provision systems in the presence of external constraints, such as a limited budget, making appropriate trade-offs to maximize benefits while limiting the system's purchase cost. The third case study provides an example of how changes in the cost of storage hardware can affect the optimal storage design by changing the cost/benefit trade-off — something not handled by minimum cost provisioning.

2 Background

A goal of many IT departments is to support and add value to the main activities of an organization in a cost-effective manner. This endeavor involves balancing the quality and benefits of services against the costs required to provide them. In provisioning a storage system, the administrator attempts to balance numerous objectives, including *performance* (e.g., bandwidth and latency), *data protection* (e.g., reliability and availability), *resource consumption* (e.g., capacity utilization and power consumption), *configuration manageability* (e.g., configuration stability and design simplicity), and *system cost*. The current best practices for this problem are

based on simple "rules of thumb" that create classes of storage to implement different points in this rich design space [31]. For example, an administrator may define storage classes, such as "business critical" or "archival," to denote particular data encodings (e.g., RAID level) and pools of disks to use for storing the data. The "business critical" class may require both high performance and a high level of reliability, while "archival" data could be placed on lower performance, lower cost, yet still reliable storage.

The task of the administrator is made more difficult with the emergence of cluster-based storage that provides the opportunity for a wider range of data placement and encoding options. For instance, both FAB [29] and Ursa Minor [1] allow the use of arbitrary m-of-n erasure codes for data protection.[1] The PASIS protocol [17] used by Ursa Minor provides the ability not only to use erasure coding to store data, but it allows the encoded data fragments to be stored onto an arbitrary subset, l, of the system's storage nodes. For example, with a "2-of-3 declustered across 4" encoding, a data object could be stored using a 2-of-3 scheme with the data spread across storage nodes one, four, five, and six.

It is difficult to determine which data encodings cost-effectively produce the proper mix of storage objectives for a particular class of data. A vendor's support organization may provide some guidance based on other installations, but the decision is ultimately up to the administrator. Unfortunately, the system administrator is ill-equipped to make this decision. He may understand that increasing the difference between m and n will provide higher reliability and potentially lower performance, but quantifying these metrics is difficult, even for experts in the field. Table 1 illustrates the general effects that changes to the data encoding parameters cause. A change to any encoding parameter affects nearly all of the system metrics — some for the better and some for the worse.

Gelb [16], Borowsky et al. [10], and Wilkes [37] have argued that administrators should be concerned with high-level system metrics, not the underlying mechanisms (e.g., the levels of performance and data protection, not the values of m, n, and l) and that a management system should automatically choose the mechanisms that produce the desired level of each metric. Unfortunately, the settings that maximize one metric are likely to severely impair others. For example, configurations that maximize reliability tend to consume a large amount of capacity, raising the hardware cost and sacrificing some performance. This lack of independence brings the need to arbitrate between, or trade off, one metric (e.g., power)

---

[1] With m-of-n erasure codes, a data block is divided into n fragments, any m of which can be used to reconstruct the original data.

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

Table 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 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.

against another (e.g., request latency). We show that utility functions provide a good method for administrators to provide automation tools with the necessary information to make these choices.

2.1 Utility

Utility is a value that represents the desirability of a particular state or outcome. This concept is common in both economics (to explain consumer preferences) and in decision theory [20] (as a method for weighing alternatives). The main feature we use in this paper is its ability to collapse multiple objectives (e.g., performance and reliability) into a single axis that can be used to compare alternatives. When presented with a suitable utility function, an automated tool can use the utility values to compare storage designs in a manner consistent with the desires of the system administrator.

To use utility to guide storage provisioning, 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 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 as the utility value. For example, setting *Utility = Bandwidth* would cause the provisioning system to prefer configurations with high bandwidth over those with low bandwidth. The goal of utility, how-

ever, 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. This restriction requires the metrics to 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 impacts an organization's business, and 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 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 easily combined.

System administrators can create utility functions by assessing the objectives of the storage system from a business perspective. This type of analysis tends to yield a set of functions related to different aspects of the storage service. For example, one function may express e-commerce revenue as a function of performance and availability. Another could describe the costs associated with a data-loss event. Other, more direct costs can be incorporated as well, such as the cost of power and cooling for the storage system or its purchase cost. These separate functions, expressed in the same "units" of currency can be summed to produce a single utility function for use in automated provisioning. While some may question the ability of the administrator to evaluate their storage needs in such business terms, we believe this approach is just providing a more formal framework for an analysis that is already performed. The system administrator is already required to justify purchase and configuration decisions to upper-level management — a conversation that is surely framed in a business context.

2.2 Related work

The problem of replica placement has been studied extensively in the context of the File Assignment Problem [12]. A similar problem has been examined for placing replicas in content distribution networks (e.g., Baev and Rajaraman [9], Tang and Xu [32]). While some of the work on FAP has examined detailed storage performance models [38], the content distribution work is largely concerned with communication costs and networks where constraints are linear, a condition that does not hold for either the performance models nor the utility functions. Additionally, because data encodings, in addition to locations, are selected as part of the optimization,

replica placement is just one part of the overall provisioning task.

Storage provisioning and configuration tools, including Minerva [2], Ergastulum [5], and the Disk Array Designer [7], have largely been targeted at creating minimum cost designs that satisfy some fixed level of performance and data protection. Our work builds on these tools by removing the fixed requirements, and, instead, using utility as the objective function. Using utility allows our system to make automatic trade-offs across the various storage metrics.
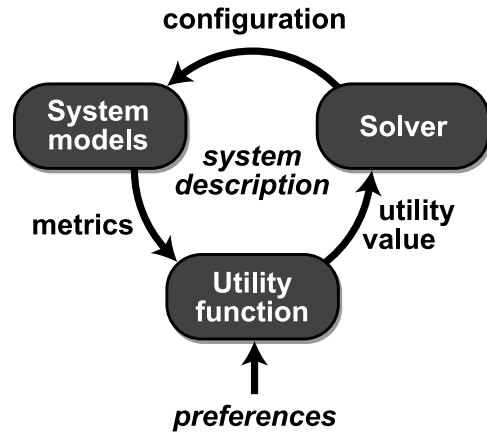
In the push toward automation, the notion of using utility to guide self-tuning and autonomic systems is becoming more popular. Kephart and Walsh [23] provide a comparison of event-condition-action (ECA) rules, goals, and utility functions for guiding autonomic systems. They note that both goals and utility are above the level of the individual system mechanisms and that utility provides a level of detail over goal-based specification that allows conflicting objectives to be reconciled automatically.

Utility has been applied to scheduling batch compute jobs to maximize usefulness for the end user in the face of deadlines [19] or where the results of many dependent jobs are needed [8]. For web-based workloads, Walsh et al. [35] describe a system that allocates server resources to control the response time between two different service classes.

There has also been work on designing cost-effective disaster recovery solutions, trading off solution costs with expected penalties for data loss and downtime [14, 21, 22]. 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 recovery. This result lends support to the notion of using business costs as the basis for evaluating storage solutions.

3   Provisioning with utility

Provisioning a storage system to provide the most value for its owner requires the ability to make appropriate trade-offs among competing objectives. A provisioning tool begins with an initial *system description* that contains the parameters of the problem, including the available hardware types as well as the workload and dataset descriptions. The tool needs three main components to automatically create cost-effective storage systems using utility. The *system models* analyze a candidate configuration, annotating it with one or more metrics. The *utility function* uses these metrics to evaluate the configuration, assigning it a single utility value that indicates the de-

Figure 2: Overview of a utility-based provisioning tool – 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.

sirability of the configuration. The *solver* generates new candidate configurations based on the feedback provided by these utility values. Figure 2 shows the interaction between these three main components.

3.1   System description

The system description provides the baseline components, such as the clients, workloads, datasets, and storage nodes, that are available to the provisioning system. These components are the building-blocks that the provisioning tool uses when creating solutions. The workloads describe the demands placed on stored data by applications. Workloads are statically assigned to clients, and their I/O requests are directed at specific datasets. The main task of the provisioning tool is to assign datasets to storage nodes, choosing the data distribution that maximizes utility. A *candidate configuration* describes the mapping of each dataset onto the storage nodes. This configuration information, combined with the system description defines a provisioned storage system that can be evaluated by the system models.

3.2   System models

System models translate from the low-level mechanism-centric configuration into a description that contains high-level system metrics, such as the performance or data protection characteristics that the design is expected to produce. For example, the configuration presented to

the system models may indicate that a dataset is encoded as "2-of-3 spread across storage nodes one, four, six, and seven." An availability model may translate this into a metric that states "the dataset has a fractional availability of 0.9997." Numerous projects have produced storage system models for performance [4, 25, 33, 34], data protection [3, 11, 15], and power [36, 39] that could potentially be used as modules.

3.3 Utility function

Using a utility function, administrators can communicate the cost and benefit structure of their environment to the provisioning tool. This allows the tool to make design trade-offs that increase the value of the system in their specific environment, "solving for" the most cost-effective level of each metric. The utility function is a mathematical expression that uses one or more of the system metrics to rank potential configurations. The function serves to collapse the many axes of interest to an administrator (the system metrics) into a single utility value that can be used by the provisioning tool.

While the function can be any arbitrary expression based on the system metrics, the purpose is to generate configurations that match well with the environment in which the system will be deployed. This argues for an approach that uses business costs and benefits as the basis for the utility function. Typically, it takes the form of a number of independent sub-expressions that are summed to form the final utility value. For example, an online retailer may derive a large fraction of its income from its transaction processing workload. Based on the average order amount, the fraction of transactions that are for new orders, and the average number of I/Os per transaction, the retailer may determine that, on average, its OLTP workload generates 0.1¢ per I/O. This would lead to an expression for the annualized revenue such as:

$$Revenue = \$0.001 \cdot IOPS_{WL} \cdot AV_{DS} \cdot \left(\frac{3.2 \times 10^7 \text{ s}}{1 \text{ yr}} \right)$$

Here, the revenue is tied to the throughput of the workload (in I/Os per second). It is also scaled by the fractional availability of the dataset because revenue is not generated when the dataset is unavailable. Finally, it is converted to an annualized amount.

The administrator would also want to add expressions for the annualized cost of repair and lost productivity during downtime (e.g., $10,000$ per hour):

$$Cost_{downtime} = \left(\frac{\$10,000}{\text{hr}} \right) \cdot (1 - AV_{DS}) \cdot \left(\frac{8766 \text{ hr}}{1 \text{ yr}} \right)$$

The cost of losing the dataset (e.g., $100 M) would be scaled by the annual failure rate:

$$Cost_{dataloss} = \$100 \text{ M} \cdot AFR_{DS}$$

These individual expressions of annual revenue and costs would be combined to form the utility function:

$$Utility = Revenue - Cost_{downtime} - Cost_{dataloss}$$

While this example has examined only a single workload and dataset, functions for other workloads and datasets in the same system can be included. In the case of independent applications (i.e., their associated revenue and costs are independent of other applications), such as combining an e-mail system with the e-commerce application, the components of utility from the different applications could be summed to produce a single composite function for the entire system. For environments where multiple applications depend on each other, a simple summation of the utility functions (i.e., assuming additive independence) may not be appropriate. Instead, the combination can be expressed by examining the benefits and costs of the combined service. For example, a web server and a database may be dependent, leading to an expression for the overall web application's cost of downtime as a function of the availability of both the database and the web server (i.e., they must both be available to provide service).

The choice to use annualized amounts is arbitrary (e.g., hourly or monthly rates could be used as well), but all utility expressions need to share the same time frame to ensure they are scaled accurately relative to each other.

The benefit of taking a business-cost approach is that the utility function can be derived from an analysis of the business' needs and objectives, reducing the need to invent a set of arbitrary requirements. While it still requires a thorough understanding of the application (e.g., to produce an estimate of the revenue per I/O), there is potential for the application, or its management tool, to assist. Future "utility-aware" applications could translate between their own high-level metrics and those of the storage system, again moving the level of specification closer to the administrator. For example, a database could provide a model for its expected transactions per second as a function of storage system performance metrics, allowing an administrator to express revenue as a function of the transactions per second achieved by the database.

3.4 Solver

The purpose of the solver is to generate improved storage system configurations based on the feedback provided

by the utility function. The solver is attempting to optimize a bin-packing problem wherein it must assign the datasets to storage nodes while attempting to maximize the administrator-provided utility function.

Developing an efficient solver can be difficult because the value (utility) of the storage system is only indirectly connected to the raw configuration settings (e.g., m, n, and l) the solver manipulates. The effect of the configuration on the utility value passes through both the system models and the utility function. Because this utility function is supplied by the system administrator at runtime, the tool designer cannot know how a configuration change is likely to affect utility, complicating the process of finding an efficient optimization algorithm.

4   A utility-based provisioning tool

We have implemented a utility-based provisioning tool that is targeted toward a cluster-based storage architecture [1] in which each client communicates directly to individual storage nodes. Datasets are spread across storage nodes using m-of-n data encodings, and the n data fragments may be declustered across an arbitrary set of l storage nodes. Workloads are statically assigned to a client, and they target a single dataset. However, clients may run multiple workloads, and datasets may be accessed by multiple workloads.

The system is described by the set of clients and workloads, the datasets they access, and the storage nodes that are available for use by the tool. Each component is described by a set of attributes specific to that component type. For example, storage nodes have attributes that describe their raw capacity, disk positioning time, streaming bandwidth, as well as network latency and bandwidth. Table 2 lists each component type and the attributes that are used to describe it.

A candidate configuration describes the mapping of each dataset onto storage nodes. The mapping is a tuple, $\{dataset, m, n, \text{list}\{storage\ nodes\}\}$, for each dataset in the system description, and the union of the storage node lists across these tuples determines the set of storage nodes in the configuration.

Our prototype tool is implemented in approximately 5000 lines of Perl. Text configuration files are used to define the characteristics of the system components and the utility function. The tool is designed to work with heterogeneous components, allowing each client, workload, storage node, and dataset to be unique.

The remainder of this section describes the tool's models and metrics, how utility is specified, and the implementation of the solver. It concludes with an evaluation of the efficiency of the tool.

– Attributes of Components –

Client:
• CPU time for data encode/decode (s)
• Network streaming bandwidth (MB/s)
• Network latency (s)
Dataset:
• Size of the dataset (MB)
Storage node:
• Annual failure rate (%)
• Fractional availability of the node (%)
• Disk capacity (MB)
• Purchase cost ($)
• Max streaming bandwidth (MB/s)
• Initial positioning time (s)
• Network streaming bandwidth (MB/s)
• Network latency (s)
• Power consumption (W)
Workload:
• Average request size (kB)
• Multi-programming level for closed-loop workload
• Think time for closed-loop workload (s)
• Fraction of non-sequential I/Os (%)
• Fraction of I/Os that are reads (%)

Table 2: Main components and their attributes – This table lists each of the main component types used in the system description for the provisioning tool. Listed with each of the component types is the set of attributes that define its 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.

4.1   Models and metrics

The system models are plug-in modules that examine a configuration and produce metrics that describe the expected characteristics and level of service the system would provide. Additional models can be easily added to the existing list, expanding the menu of metrics that can be used by the utility function. The discussion below describes the existing set of models and the metrics that they provide. These models cover a wide range in their complexity and analysis techniques, highlighting the versatility of this architecture. While there exist potentially more accurate models for each of these metrics, the existing models provide sufficient detail to evaluate the value of utility for storage provisioning.

Performance: The performance model is the most intricate of the current models. It provides utilization

estimates for disk and network resources as well as throughput and latency estimates for the workloads. This information is derived from a closed-loop queueing model that is constructed and solved based on the system description and dataset assignments in the candidate configuration. The model is similar to that described by Thereska et al. [33], with queueing centers for the clients' CPU and network as well as the storage nodes' network and disk. The service demand placed on each queueing center is based on the encoding parameters for each dataset as well as the read/write ratio, I/O size, and sequentiality of the workloads. Workload intensity is specified as a multiprogramming level and a think time. These correspond to a maximum level of parallelism and an application's processing delay between I/Os, respectively. The open source PDQ [18] queueing model solver is used to analyze the model via Mean Value Analysis.

Availability: The availability model estimates the fractional availability of each dataset based on the availability of the storage nodes that the dataset is spread across and the encoding parameters used. The module calculates the dataset's availability as:
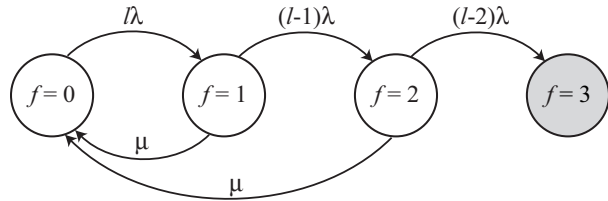
$$AV = \sum_{f=0}^{n-m} \binom{l}{l-f} A_{SN}^{l-f} (1 - A_{SN})^f$$

where n and m are the encoding parameters for the dataset, l is the number of storage nodes that the data is spread across, and A_{SN} is the minimum individual availability of the set of l storage nodes. The dataset is considered to be available as long as no more than $n - m$ of the l storage nodes are down.[2] The formula above sums the probability for each "available" state ($f = 0 \ldots (n-m)$).

This model makes some simplifications, such as using a single availability value for all nodes. It also assumes independent failures of storage nodes, an assumption that has been called into question [28, 30] but is nonetheless sufficient for this study.

Reliability: The reliability model uses a Markov chain to estimate the annual failure rate for a dataset. The chain has $n - m + 2$ states, representing the number of device failures. The final state of the chain is an absorbing state, representing a data loss event. The transition rates for device failures are calculated as the number of storage nodes that contain fragments for the dataset (l) times the failure rate of the individual nodes. In the case where nodes have differing failure rates, the maximum is used, producing a conservative estimate. Repair operations are handled by re-encoding the dataset. The re-encode operation repairs all failures in the same operation, caus-

---

[2]We focus on a synchronous timing model and a crash failure model for the storage nodes.

Figure 3: Markov chain for a 1-of-3 data encoding – A 1-of-3 data encoding is able to withstand up to 2 failures without losing data; the third failure ($f = 3$) results in a data loss. The transition rates between states are, in the case of failure transitions, related to the number of storage nodes that hold data for this dataset, l, and the annual failure rate of the nodes. The repair rate is calculated as a fixed fraction of the individual storage nodes streaming bandwidth (e.g., 5% of the slowest node) and the size of the dataset. A single repair operation is able to fix multiple failures simultaneously.

ing all repair transitions to lead to the failure-free state. The repair rate is based on the size of the dataset and a fixed fraction of the streaming bandwidth from the slowest storage node in the data distribution. The chain is solved for the expected time until the absorbing state is reached using the technique described by Pâris et al. [27]. Figure 3 shows an example chain for a 1-of-3 encoding. This reliability model represents only the likelihood of losing data from the primary storage system (i.e., it does not account for remote mirroring or external backup). More detailed models incorporating backup and remote replication, such as those by Keeton et al. [21], could be used.

Capacity: The capacity model calculates the storage blowup of each dataset due to the redundancy of the encoding scheme. The storage blowup is calculated as: $\frac{n}{m}$. It also calculates the capacity usage of each storage node based on the datasets they store.

Cost and power: The purchase cost and power models are very similar, producing system-wide metrics based on the quantity and type of storage nodes that are used. Each storage node type has fixed cost and power attributes. When a storage node is used in a configuration, it is assumed to cost a fixed amount to purchase and consume a fixed amount of power. These cost and power attributes are then summed to produce the system-wide metrics for cost and power consumption (e.g., a system with three storage nodes that cost $10 k each would have a total system cost of $30 k).

Table 3 lists the set of metrics provided by the current system models. This list provides the framework for creating utility functions to evaluate configurations. Additional models can be added to expand the menu of metrics available for the administrator to express his ob-

– System Metrics –
Client:
• CPU utilization (%)
• Network utilization (%)
Dataset:
• Annual failure rate (%)
• Fractional availability (%)
• Capacity blowup from encoding
• Mean time to failure (hr)
• "Nines" of availability
Storage node:
• Raw capacity consumed (MB)
• Capacity utilization (%)
• Disk utilization (%)
• Network utilization (%)
• Power consumed (W)
System-wide:
• Total capacity consumed (MB)
• Capacity utilization (%)
• System power consumed (W)
• Total system cost ($)
Workload:
• Bandwidth (MB/s)
• Throughput (IO/s)
• Request latency (s)

Table 3: Storage metrics provided by system models – This table lists the metrics that are calculated for candidate configurations by the current set of system models. The table is organized by the component to which the metric refers. With the exception of the system-wide metrics, these metrics are calculated for each instance of a component (e.g., each storage node has a disk utilization that is calculated based on the accesses that it receives).

jectives. For example, an additional availability model could be created that estimates the frequency and duration of outages instead of just a fractional availability.

4.2 Specifying utility

The provisioning tool's interface for specifying utility allows the use of an arbitrary function that assesses the metrics for the current system configuration. The utility function returns a single floating point number that is the utility value for the configuration. It is specified in the text configuration file as a block of Perl syntax that is `eval()`-ed when the configuration is loaded. This code has the ability to use any of the metrics listed in Table 3 when computing utility. Maintaining such a flexible interface to specify utility has proven valuable for experimentation. It allows not only utility functions based on business costs (as discussed in Section 3.3) but also util-

ity functions that implement strict priorities (e.g., first obtain 4 "nines" of availability, next achieve 300 IO/s, then minimize capacity utilization).

4.3 Solver

While many optimization techniques could be employed to generate candidate configurations, we have chosen to use a solver based on a genetic algorithm [26]. It refines a population of candidate solutions over a number of generations. Each generation contains a population of 100 candidates that are evaluated by the models and utility function. The utility value is used to generate a *fitness* value for each candidate. These fitness values are used to create a new population of candidates, and the process repeats. The creation of a new population based on the existing population occurs via *selection*, *crossover*, and *mutation* operations. These operations each introduce randomness into the search, attempting to avoid local maxima and maintain diversity within the population. The solver continues, producing configurations with higher utility, until some stopping condition is reached. As the solver progresses through a number of generations, the observed gains in utility diminish. The solver terminates if there has been no improvement upon the best configuration for 40 generations.

Fitness: The fitness value determines how likely a candidate is to be selected for reproduction into the next generation. The fitness value 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, fitness is calculated as:

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

OCC is the sum of the over-committed capacity from the individual storage nodes (in MB). For those infeasible solutions, the fitness value will be between zero and one. When the solution is feasible, $OCC = 0$ (no storage nodes are over-committed). Utility values less than two are compressed into the range between two and one, eliminating negative utility to be compatible with the Roulette selection algorithm and ensuring that all feasible solutions have a fitness value greater than the infeasible ones. This type of penalty scheme for infeasible solutions is similar to that used by Feltl and Raidl [13].

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. The solver can use either Tournament or Roulette selection

algorithms [26], but it defaults to Tournament. Empirically, the solver performs better using Tournament selection for utility functions that are ordinal (where the utility value specifies only an ordering of candidates, not "how much" better a given solution is). With Tournament selection, two candidates are chosen at random (uniformly) from the current population. From these two candidates, the one with the larger fitness value is output as the selected candidate. The result of this process is that the selection algorithm chooses candidates weighted by their rank within the current population.

Roulette selection, on the other hand, chooses candidates for the next generation with a probability proportional to the relative magnitude of their fitness values. For example, a candidate with a fitness value of 100 would be twice as likely to be selected as one with a value of 50.

Candidate representation: For use in the genetic algorithm (GA), the storage configuration space is encoded as a matrix. This matrix has one row for each dataset and one column for each storage node. Each location in the matrix may take on integer values between zero and three, inclusive. Any non-zero value at a particular location is used to indicate that the dataset (represented by the row) is stored on the storage node (represented by the column). The values of the m and n encoding parameters for a dataset are determined by 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 1 2] 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: $1 \leq m \leq n \leq l$. This representation ensures the relationship of m, n, and l. The only condition that must be verified and corrected is that 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.

When the GA is initialized (i.e., the first generation is created), the matrix is generated randomly, with values from zero to three in each cell. These values are then changed via the crossover and mutation operations as the optimization progresses.

Crossover operator: 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). The solver uses *uniform crossover* to achieve this mixing. In uniform crossover, each "gene" has an independent 50% probability of coming from either original candidate. Crossover is performed at the "dataset" level — an entire row of the configuration matrix is selected as a single unit. This approach was chosen because the individual values within a row have little meaning when taken independently, and combining at the dataset level is more likely to form a new set of candidates with properties of the original two.

Mutation operator: The mutation operator is used to add randomness to the search. It operates on a single candidate at a time by changing a random location in the configuration matrix to a new random integer in the range of zero to three. Before a particular value is changed, it is verified that the new value will not cause the configuration to be invalid — a location with a current value of "3" can only be changed if there is at least one other "3" in that row. If a conflict is found, a different location is chosen for mutation.

The GA also supports being used for "incremental" provisioning, wherein additional datasets, and potentially hardware, are added to an existing system. In this case, the existing datasets can be held constant by omitting them from the matrix representation, only using them when utility is calculated. Evaluating whether an existing dataset should be reassigned as a part of the incremental provisioning process is left as an area of future work.
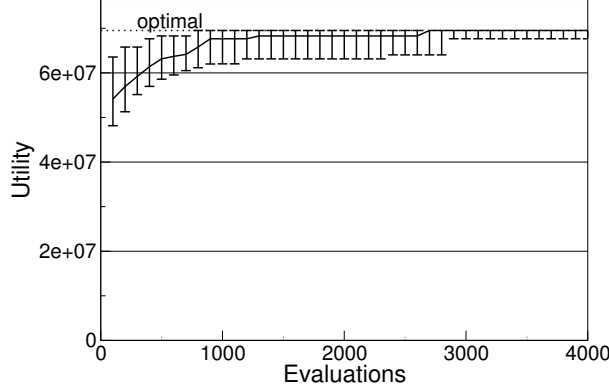
4.4 Tool effectiveness

For utility to be useful at guiding the provisioning of storage, it must be possible to reliably find configurations that have high (near optimal) utility. In the following experiments, we show that the solutions produced by the solver approach the optimum and that the solver is able to quickly find good solutions for difficult problems. Although better algorithms likely exist, these experiments demonstrate that it is possible to create an effective utility-based provisioning tool.

4.4.1 Convergence toward the optimum

This experiment compares the solutions produced by the genetic solver with the optimal solution produced by an exhaustive search of the configuration space. To constrain the problem so that it can be solved by exhaustive search, configurations may use a maximum of eight storage nodes.

The utility function used for this scenario is identical to the example presented in Section 3.3 but presented as

Figure 4: 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.

utility (costs are represented as negative utility):

$$Utility = U_{revenue} + U_{dataloss} + U_{downtime}$$

$$U_{revenue} = \$0.001 \cdot AV_{DS} \cdot IOPS_{WL} \cdot \left(\frac{3.2 \times 10^7 \text{ s}}{1 \text{ yr}}\right)$$

$$U_{dataloss} = -\$100 \text{ M} \cdot AFR_{DS}$$

$$U_{downtime} = \left(\frac{-\$10,000}{\text{hr}}\right) \cdot (1 - AV_{DS}) \cdot \left(\frac{8766 \text{ hr}}{1 \text{ yr}}\right)$$

For this experiment, there are two identical clients, each with one workload, issuing I/O requests to separate datasets. The total utility is the sum across both of the workloads and datasets. The simulated storage nodes have 5.5 ms average latency and 70 MB/s max streaming bandwidth from their disk. They are assumed to have an individual availability of 0.95 and an annual failure rate of 1.5%.

Figure 4 shows the results for this experiment. The exhaustive solution produces a utility of 6.95×10^7, 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 7.9×10^6 total configurations are within 10% of the optimum. The utility values across the configuration space range from -7.23×10^7 to 6.95×10^7.

4.4.2 Finding rare solutions

The previous experiment provided an example showing that the solver quickly approaches the optimal solution. This experiment will explore how well the solver is able to find rare solutions. The ability to find rare solutions is important because some combinations of workloads, hardware, and utility functions have the characteristic that there are very few configurations within a small percentage of the optimal solution.
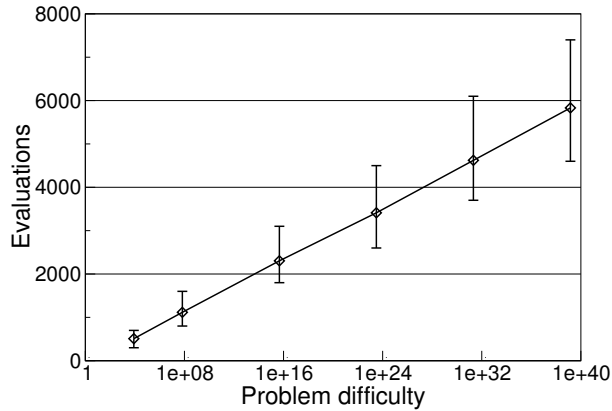
For this experiment, the same storage nodes are used, but the number of clients, datasets, and workloads are scaled together with a 1:1:1 ratio to control the problem's difficulty. A (contrived) utility function is constructed so that it is possible to predict the number of "desirable" configurations as a fraction of the total number of possible storage configurations. For this experiment, the definition of "desirable" is that all datasets 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. 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, $\left(\frac{464}{2816}\right)^3 = .4\%$ of the possible configurations have 4 nines for all three workloads.

The utility function for this scenario is:

$$U = \frac{1}{S} \cdot \sum_{i=1}^{S} \min(NINES_{DS_i}, 4)$$

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 the storage nodes' capacity is chosen to ensure the system is not capacity constrained.

Figure 5 shows how the 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 "desirable" solutions result in an approximately linear growth in the number of configurations that must be evaluated by the solver.

Figure 5: Scalability of genetic solver – 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 before finding a valid solution. The error bars indicate the 5th and 95th percentiles.
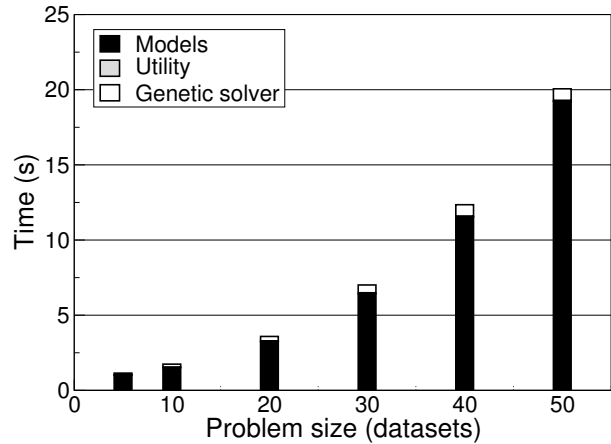
4.4.3 Solution speed

Our provisioning tool, implemented in Perl, solves the above "4 nines" scenarios in a few minutes. 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.

Figure 6 shows the average time required to evaluate one generation of 100 candidates for each of the problem sizes. It divides this per-generation time into three categories, corresponding to the three main system components (models, utility function, and genetic solver). The majority of the runtime (and growth in runtime) is consumed by the system models, suggesting that efficient (but still accurate) models are a key to effective utility-based provisioning. The time required to calculate the utility value from the system metrics is too small to be visible on the graph.

5 Case studies

The benefits of using utility as the basis for storage provisioning can be seen by examining several case studies. This section uses our utility-based provisioning tool to explore three different scenarios, highlighting several important benefits of using utility to evaluate designs.

The simulated system components used for the case studies are described in Table 4. The same workload descrip-

Figure 6: Speed of genetic solver – This graph shows how the evaluation time changes as a function of the problem size. The x-axis is the number of datasets being optimized. There is one client and workload for each. The y-axis is the time required to fully evaluate one generation of candidate solutions (100 configurations). The system models consume the majority of the processing time, and the actual evaluation of the utility function is insignificant.

tion is used for the first two case studies, however the third uses a workload description more appropriate to the described application (trace processing).

5.1 Performance vs. availability

Provisioning and configuration decisions affect multiple system metrics simultaneously; nearly every choice involves a trade-off. By 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 involved 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. In this example, the cost of electricity to power the system was also added at a cost of $0.12 per kWh, and the purchase cost of the system (storage nodes) was amor-

Client			
CPU	0.2 ms	Net latency	125 μs
Net bw	119 MB/s		
Dataset			
Size	100 GB		
Storage node			
AFR	0.015	Avail	0.95
Capacity	500 GB	Cost	$5000
Disk bw	70 MB/s	Disk latency	5.5 ms
Net bw	119 MB/s	Net latency	125 μs
Power	50 W		
Workload (§5.1, §5.2)			
I/O size	8 kB	MP level	5
Think time	1 ms	Rand frac	0.5
Read frac	0.5		
Workload (§5.3)			
I/O size	32 kB	MP level	10
Think time	1 ms	Rand frac	0.0
Read frac	1.0		

Table 4: Components used for case studies – This table lists the main attributes of the components that are used as a basis for the case study examples. The client is based on measurements from a 2.66 GHz Pentium 4. The storage node data is based on the data sheet specifications for a single disk drive, combined with a 1 Gb/s network connection, processing and cache.

tized over a three year expected lifetime:

$$Utility = U_{perf} + U_{avail} + U_{rel} + U_{power} + U_{cost}$$

$$U_{perf} = (\text{see below})$$

$$U_{avail} = \left(\frac{-\$10,000}{\text{hr}} \right) \cdot (1 - AV_{DS}) \cdot \left(\frac{8766 \text{ hr}}{1 \text{ yr}} \right)$$

$$U_{rel} = -\$100 \text{ M} \cdot AFR_{DS}$$

$$U_{power} = \left(\frac{-\$0.12}{\text{kW·hr}} \right) \cdot Power \cdot \left(\frac{8766 \text{ hr}}{1 \text{ yr}} \right) \cdot \left(\frac{1 \text{ kW}}{1000 \text{ W}} \right)$$

$$U_{cost} = \left(\frac{-Cost}{3 \text{ yr}} \right)$$

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

$$U_{perf\ 0.1} = \$0.001 \cdot AV_{DS} \cdot IOPS_{WL} \cdot \left(\frac{3.2 \times 10^7 \text{ s}}{1 \text{ yr}} \right)$$

$$U_{perf\ 0.01} = \$0.0001 \cdot AV_{DS} \cdot IOPS_{WL} \cdot \left(\frac{3.2 \times 10^7 \text{ s}}{1 \text{ yr}} \right)$$

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 5 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: 1-of-2 declustered across 6 (1/2/6) for the 0.1¢ scenario and 1-of-3 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.

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 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 in 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. However, 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.

5.2 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. 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 the best option that fits within his budget.

Using the previous 0.01¢ scenario 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 compares this optimal solution to two alternatives that have constraints on the total cost of the storage hardware. The first alternative limits the total hardware budget to $30 k, and the second further reduces it to $20 k. Notice that these two alternatives use six and four storage nodes respectively (at $5000 each) to stay within their budget. The "AP cost" in the table reflects this cost amortized 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 the optimum ($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 the optimum leads to a 10% loss of utility, as performance is significantly impacted.

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.

5.3 Price sensitivity

Even without budgetary constraints, the price of the storage hardware can impact the proper solution. Consider the case of an academic research group whose students process file system traces as a part of their daily work. The trace processing application reads a trace (27 GB on

	$70 k	$30 k	$20 k
Total 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
AP cost	−$23 k	−$10 k	−$6.6 k
Total utility	$6.2 M	$6.1 M	$5.6 M

Table 6: **Utility can be used to design for limited storage budgets** – The optimal system configuration costs a total of $70 k, giving an amortized purchase (AP) cost of $23 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.

average in this scenario) sequentially and generates a set of summary statistics. Assuming that the students cost $35 per hour[3], that they wait for the results of a run, and that there are 250 runs per year (approximately one for each regular workday), the total cost incurred is:

$$U_{perf} = \left(\frac{-\$35}{hr}\right) \cdot \left(\frac{27 \text{ GB}}{run}\right) \cdot \frac{1}{BW_{WL}} \cdot \left(\frac{250 \text{ runs}}{yr}\right)$$
$$\cdot \left(\frac{hr}{3600 \text{ s}}\right) \cdot \left(\frac{1024 \text{ MB}}{GB}\right)$$
$$= \frac{67200}{BW_{WL}}$$

If the traces are lost from primary storage, it is projected to require 15 hours of administrator time (at $35 per hour) to re-acquire and restore the traces:

$$U_{rel} = \left(\frac{-\$35}{hr}\right) \cdot (15 \text{ hr}) \cdot AFR_{DS}$$

---

[3]This amount is the cost to the research program, not what the students are paid, unfortunately.

	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	2.0×10^{-6} afr	1.1×10^{-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**

Table 5: **Effect of workload importance on provisioning decisions** – 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*.

	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
AP cost	−\$6667	−\$1333	−\$2000
Total utility	−\$9180/yr	−\$3846/yr	−\$3097/yr

Table 7: **The price of the storage hardware affects the optimal storage configuration** – The optimal configuration using "expensive" (\$10 k each) storage nodes is two-way mirroring. However, 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 annual costs for the storage system.

If the traces are unavailable, the administrator and one student will be occupied troubleshooting and fixing the problem:

$$U_{avail} = \left(\frac{-\$70}{\text{hr}} \right) \cdot (1 - AV_{DS}) \cdot \left(\frac{8766 \text{ hr}}{1 \text{ yr}} \right)$$

The storage system must be powered, and the purchase cost will be spread across a three year lifetime:

$$U_{power} = \left(\frac{-\$0.12}{\text{kW·hr}} \right) \cdot Power \cdot \left(\frac{8766 \text{ hr}}{1 \text{ yr}} \right) \cdot \left(\frac{1 \text{ kW}}{1000 \text{ W}} \right)$$

$$U_{cost} = \left(\frac{-Cost}{3 \text{ yr}} \right)$$

Provisioning this system using storage nodes as described in Table 4 but using a total cost of \$10 k per node (\$3.3 k annualized) leads to a solution using two storage nodes and 2-way mirroring. The first column of Table 7 shows a breakdown of the costs using these "expensive" storage nodes. If the cost of a storage node is reduced to \$2 k each (\$667 annualized), the total costs obviously decrease due to the lower acquisition cost (second column of Table 7). 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 Conclusion

Producing cost-effective storage solutions requires balancing the costs of providing storage with the benefits that the system will provide. Choosing a proper storage configuration requires balancing many competing system metrics in the context where the system will be deployed. Using utility, it is possible to bring these costs and benefits into the same framework, allowing an automated tool to identify cost-effective solutions that meet identified constraints.

This paper illustrates the value of utility-based provisioning with three case studies. The first case study shows an example where trade-offs exist between metrics, such as performance and availability, and utility provides a method to navigate them. The second shows that utility functions are flexible enough to be used in the presence of external constraints, such as a limited budget. The third shows that provisioning a storage system is not just limited to finding the "minimum cost" solution that meets a set of requirements, because the system cost can have a impact on the solution.

This investigation shows the potential for using utility to guide static storage provisioning. By analyzing utility over time, there is also the potential to provide guidance for automated, online tuning as well. For example, a modified version of our provisioning tool could be used to generate new candidate configurations, evaluate long-term expected utility, and decide whether the change is advantageous and how fast to migrate or re-encode the data. Exploring such on-line use of utility is an interesting area for continuing work.

7 Acknowledgements

References

[1] M. Abd-El-Malek, W. V. Courtright II, C. Cranor, G. R. Ganger, J. Hendricks, A. J. Klosterman, M. Mesnier, M. Prasad, B. Salmon, R. R. Sambasivan, S. Sinnamohideen, J. D. Strunk, E. Thereska, M. Wachs, and J. J. Wylie. Ursa Minor: Versatile cluster-based storage. In *Conference on File and Storage Technologies*, pages 59–72. USENIX Association, 2005. ISBN 978–1–931971–39–3.

[2] G. A. Alvarez, J. Wilkes, E. Borowsky, S. Go, T. H. Romer, R. Becker-Szendy, R. Golding, A. Merchant, M. Spasojevic, and A. Veitch. Minerva: An automated resource provisioning tool for large-scale storage systems. *ACM Transactions on Computer Systems*, 19(4):483–518, November 2001. ISSN 0734–2071. doi: 10.1145/502912.502915.

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

[4] E. Anderson. Simple table-based modeling of storage devices. Technical Report HPL–SSP–2001–4, Hewlett-Packard Laboratories, July 2001.

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

[6] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. Veitch. Hippodrome: Running circles around storage administration. In *Conference on File and Storage Technologies*, pages 175–188. USENIX Association, 2002. ISBN 978–1–880446–03–4.

[7] E. Anderson, S. Spence, R. Swaminathan, M. Kallahalla, and Q. Wang. Quickly finding near-optimal storage designs. *ACM Transactions on Computer Systems*, 23(4):337–374, November 2005. ISSN 0734–2071. doi: 10.1145/1113574.1113575.

[8] A. AuYoung, L. Grit, J. Wiener, and J. Wilkes. Service contracts and aggregate utility functions. In *International Symposium on High-Performance Distributed Computing*, pages 119–131. IEEE, 2006. ISBN 978–1–4244–0307–3. doi: 10.1109/HPDC.2006.1652143.

[9] I. D. Baev and R. Rajaraman. Approximation algorithms for data placement in arbitrary networks. In *Symposium on Discrete Algorithms*, pages 661–670. Society for Industrial and Applied Mathematics, 2001. ISBN 978–0–89871–490–6.

[10] E. Borowsky, R. Golding, A. Merchant, L. Schreier, E. Shriver, M. Spasojevic, and J. Wilkes. Using attribute-managed storage to achieve QoS. In *International Workshop on Quality of Service*, pages 203–207. IFIP, 1997. ISBN 978–0–412–80940–8.

[11] W. A. Burkhard and J. Menon. Disk array storage system reliability. In *Symposium on Fault-Tolerant Computer Systems*, pages 432–441. IEEE, 1993. ISBN 978–0–8186–3680–6. doi: 10.1109/FTCS.1993.627346.

[12] L. W. Dowdy and D. V. Foster. Comparative models of the file assignment problem. *ACM Computing Surveys*, 14(2):287–313, June 1982. ISSN 0360–0300. doi: 10.1145/356876.356883.

[13] H. Feltl and G. R. Raidl. An improved hybrid genetic algorithm for the generalized assignment problem. In *Symposium on Applied Computing*, pages 990–995. ACM Press, 2004. ISBN 978–1–58113–812–2. doi: 10.1145/967900.968102.

[14] S. Gaonkar, K. Keeton, A. Merchant, and W. H. Sanders. Designing dependable storage solutions for shared application environments. In *International Conference on Dependable Systems and Networks*, pages 371–382. IEEE Computer Society, 2006. ISBN 978–0–7695–2607–2. doi: 10.1109/DSN.2006.27.

[15] R. Geist and K. Trivedi. An analytic treatment of the reliability and performance of mirrored disk subsystems. In *Symposium on Fault-Tolerant Computer Systems*, pages 442–450. IEEE, 1993. ISBN 978–0–8186–3680–6. doi: 10.1109/FTCS.1993.627347.

[16] J. P. Gelb. System-managed storage. *IBM Systems Journal*, 28(1):77–103, 1989. ISSN 0018–8670.

[17] G. R. Goodson, J. J. Wylie, G. R. Ganger, and M. K. Reiter. Efficient Byzantine-tolerant erasure-coded storage. In *International Conference on Dependable Systems and Networks*, pages 135–144. IEEE Computer Society, 2004. ISBN 978–0–7695–2052–0. doi: 10.1109/DSN.2004.1311884.

[18] N. Gunther and P. Harding. PDQ (pretty damn quick) version 4.2. http://www.perfdynamics.com/Tools/PDQ.html, 2007.

[19] D. E. Irwin, L. E. Grit, and J. S. Chase. Balancing risk and reward in a market-based task service. In *International Symposium on High-Performance Distributed Computing*, pages 160–169. IEEE, 2004. ISBN 978–0–7695–2175–6. doi: 10.1109/HPDC.2004.1323519.

[20] R. L. Keeney and H. Raiffa. *Decisions with Multiple Objectives: Preferences and Value Tradeoffs*. Cambridge University Press, 1993. ISBN 978–0–521–43883–4.

[21] K. Keeton, C. Santos, D. Beyer, J. Chase, and J. Wilkes. Designing for disasters. In *Conference on File and Storage Technologies*, pages 59–72. USENIX Association, 2004. ISBN 978–1–931971–19–5.

[22] K. Keeton, D. Beyer, E. Brau, A. Merchant, C. Santos, and A. Zhang. On the road to recovery: Restoring data after disasters. In *European Systems Conference*, pages 235–248. ACM Press, 2006. doi: 10.1145/1217935.1217958.

[23] J. O. Kephart and W. E. Walsh. An artificial intelligence perspective on autonomic computing policies. In *International Workshop on Policies for Distributed Systems and Networks*, pages 3–12. IEEE, 2004. ISBN 978–0–7695–2141–1. doi: 10.1109/POLICY.2004.1309145.

[24] M. Litzkow, M. Livny, and M. Mutka. Condor - a hunter of idle workstations. In *International Conference on Distributed Computing Systems*, pages 104–111. IEEE, 1988. ISBN 978–0–8186–0865–0. doi: 10.1109/DCS.1988. 12507.

[25] M. P. Mesnier, M. Wachs, R. R. Sambasivan, A. Zheng, and G. R. Ganger. Modeling the relative fitness of storage. In *Conference on Measurement and Modeling of Computer Systems*, pages 37–48. ACM Press, 2007. doi: 10.1145/1254882.1254887.

[26] T. M. Mitchell. *Machine Learning*. McGraw-Hill, 1997. ISBN 978–0–07–042807–2.

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

[28] E. Pinheiro, W.-D. Weber, and L. A. Barroso. Failure trends in a large disk drive population. In *Conference on File and Storage Technologies*, pages 17–28. USENIX Association, 2007. ISBN 978–1–931971–50–8.

[29] Y. Saito, S. Frølund, A. Veitch, A. Merchant, and S. Spence. FAB: building distributed enterprise disk arrays from commodity components. In *Architectural Support for Programming Languages and Operating Systems*, pages 48–58. ACM Press, 2004. doi: 10.1145/ 1024393.1024400.

[30] B. Schroeder and G. A. Gibson. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you? In *Conference on File and Storage Technologies*, pages 1–16. USENIX Association, 2007. ISBN 978–1–931971–50–8.

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

[32] X. Tang and J. Xu. On replica placement for QoS-aware content distribution. In *Joint Conference of the IEEE Computer and Communications Societies*, volume 2, pages 806–815. IEEE, 2004. ISBN 978–0–7803–8355–5.

[33] E. Thereska, M. Abd-El-Malek, J. J. Wylie, D. Narayanan, and G. R. Ganger. Informed data distribution selection in a self-predicting storage system. In *International Conference on Autonomic Computing*, pages 187–198. IEEE, 2006. ISBN 978–1–4244–0175–8.

[34] E. Varki, A. Merchant, J. Xu, and X. Qiu. Issues and challenges in the performance analysis of real disk arrays. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):559–574, June 2004. ISSN 1045–9219. doi: 10. 1109/TPDS.2004.9.

[35] W. E. Walsh, G. Tesauro, J. O. Kephart, and R. Das. Utility functions in autonomic systems. In *International Conference on Autonomic Computing*, pages 70–77. IEEE, 2004. ISBN 978–0–7695–2114–5. doi: 10.1109/ICAC. 2004.1301349.

[36] C. Weddle, M. Oldham, J. Qian, A.-I. A. Wang, P. Reiher, and G. Kuenning. PARAID: A gear-shifting power-aware RAID. *ACM Transactions on Storage*, 3(3):13, October 2007. ISSN 1553–3077. doi: 10.1145/1289720.1289721.

[37] J. Wilkes. Traveling to Rome: QoS specifications for automated storage system management. In *International Workshop on Quality of Service*, pages 75–91. IFIP, 2001. ISBN 978–3–540–42217–4.

[38] J. Wolf. The Placement Optimization Program: A practical solution to the disk file assignment problem. *Performance Evaluation Review*, 17(1):1–9, May 1989. ISSN 0163–5999. doi: 10.1145/75108.75373.

[39] Q. Zhu, Z. Chen, L. Tan, Y. Zhou, K. Keeton, and J. Wilkes. Hibernator: Helping disk arrays sleep through the winter. In *Symposium on Operating System Principles*, pages 177–190. ACM Press, 2005. ISBN 978–1–59593–079–8. doi: 10.1145/1095810.1095828.