Designing dependable storage solutions for shared application environments

Shravan Gaonkar* Kimberly Keeton# Arif Merchant# William H. Sanders*
*{gaonkar,whs}@uiuc.edu, # {kimberly.keeton,arif.merchant}@hp.com
*Coordinated Science Laboratory, University of Illinois at Urbana-Champaign
#Hewlett Packard Labs, Palo Alto, California

Abstract

The costs of data loss and unavailability can be large, so businesses use many data protection techniques, such as remote mirroring, snapshots and backups, to guard against failures. Choosing an appropriate combination of techniques is difficult because there are numerous approaches for protecting data and allocating resources. Storage system designers typically use ad hoc techniques, often resulting in over-engineered, expensive solutions or under-provisioned, inadequate ones. In contrast, this paper presents a principled, automated approach for designing dependable storage solutions for multiple applications in shared environments. Our contributions include search heuristics for intelligently exploring the large design space and modeling techniques for capturing interactions between applications during recovery. Using realistic storage system requirements, we show that our design tool can produce designs that cost up to 3X less in initial outlays and expected data penalties than the designs produced by an emulated human design process.

1 Introduction

Businesses today rely critically on their IT infrastructure, and events that cause data unavailability or loss can have expensive, or even catastrophic, consequences. Such events can include natural disasters, equipment failures, software failures, user and administrator errors, and malicious attacks. Given these threats, most businesses protect their data using techniques such as remote mirroring, point-in-time copies (e.g., snapshots), and periodic backups to tape or disk. These techniques have different properties, advantages, and costs. For example, using synchronous remote mirroring permits applications to be quickly failed over and resumed at the remote location. Snapshots internal to a disk array are space-efficient and permit fast recovery of a consistent recent version of the data. Backups to tape or disk allow an older version of the data to be recovered. On the other hand, remote mirroring usually has high resource requirements, local snapshots do not protect against failure of the disk array, and recovering from backups can result in significant loss of recent updates.

To achieve adequate levels of data protection, it may be necessary to use a combination of techniques. The storage system designer must select one or more data protection techniques to apply to each application workload. Resources, such as disk arrays, servers, tape libraries and network links, must also be assigned to the application to support these techniques. The resources and data protection techniques have many configuration parameters; for example, a backup schedule needs to specify the frequency of the backups and whether the backups will be full or incremental. The designer must verify that the design will meet both normal operational performance requirements (for example, the backups will complete overnight), and that the recovery behavior will be adequate under various expected failure scenarios. These decisions must be made in a cost-effective manner. Faced with such complexity, designers usually resort to simple ad hoc heuristics: categorize applications by importance (gold, silver, or bronze) and assign a standard data protection design depending upon the category. This approach frequently results in either an over-engineered system that is more expensive than necessary, or an under-provisioned one that does not meet requirements.

In this paper, we provide a principled, automated approach to designing dependable data storage systems for multi-application environments, which minimizes the overall cost of the system while meeting business requirements. Previous work considers how to automatically design a dependable storage system that uses a single technique to protect a single application workload [11] and how to evaluate the recovery behavior of a single application workload protected by a combination of techniques [13]. Storage system design for multi-application environments presents even greater challenges. First, the design space of data protection technique and resource configurations for multiple applications is extremely large. Second, if multiple applications and their data protection techniques share the same physical resources, either in non-failure mode or in recovery mode, contention for these resources may impact application performance, compared to the case where each application operates in isolation. Third, designing for multiple applications may prompt different design decisions than if each application were considered in isolation. For instance, it may be more cost-effective to consolidate multiple workloads (even if some are less important) onto a high-end disk array than to employ a high-end array for important work-

loads and a less expensive array for less important workloads.

Our contributions include search heuristics for intelligently exploring the large design space, as well as modeling techniques for capturing interactions between applications during recovery, which extend earlier single-application models [13] to multi-application environments. We quantitatively evaluate our heuristic approach using realistic storage system environments and compare its solutions to those produced by a simple heuristic that emulates human design choices. For the scenarios we study, we find that our approach's solutions reduce overall system costs due to equipment and software outlays and data unavailability and loss penalties by at least a factor of three.

The remainder of this paper is organized as follows. Section 2 formulates the problem of designing storage systems to protect application data. In Section 3, we describe our design methods. In Section 4, we evaluate our approach quantitatively. Section 5 presents related work, and Section 6 concludes.

2 Designing dependable storage systems

Our goal is to find the best storage solution, which is the one that minimizes overall costs, including infrastructure outlays as well as penalties for application downtime and data loss. The solution to this problem specifies: 1) a combination of data protection and recovery techniques for each application workload (e.g., remote synchronous mirroring and local snapshots and local backup); 2) how those data protection techniques should be configured (e.g., how frequently snapshots and backups are taken); and 3) how physical resources like disk arrays, tape libraries and network links should be provisioned to support normal and recovery operation.

To understand how the design tool makes choices among design alternatives, this section describes the design space and all the parameters used to prescribe a particular design. We begin by describing how we model the design space, including the data protection and recovery techniques, application workload characteristics, device infrastructure, and failure scenarios. We then describe how the cost of a particular solution is computed and provide a precise description of the problem we solve, in terms of this design space.

2.1 Data protection and recovery techniques

Protecting applications against data loss and unavailability requires making one or more secondary copies of the data that can be isolated from failures of the primary data copy. Although standard redundant hardware techniques such as RAID [16] are used to protect data from internal hardware failures, they are not sufficient to protect data from other kinds of failures, such as human errors, software failures or site failure due to disasters. Geographic distribution of secondary copies (e.g., through inter-array mirroring [10, 17] or remote vaulting) provides resilience against site and regional disasters. Point-in-time [4] and backup [5, 9, 18] copies address application data object errors like accidental deletion and software failures, such as buggy software or virus infection, by permitting restoration of a previously consistent copy. These data protection techniques can be combined to provide more complete coverage for a broader set of threats.

After a failure, application data can be recovered either by restoring one of the secondary copies at the primary site or a secondary site, or by, failing over to a secondary mirror. Failover requires a later fail back operation (performed in the background) to copy data and transfer computation back to the target site.

We leverage the framework described in [13] to model data protection and recovery technique behavior, including creation, retention and propagation of secondary copies. Primary and secondary copies are modeled as a hierarchy, where each level in the hierarchy corresponds to either the primary copy or one of the techniques used to maintain a secondary copy. The data protection technique parameters specify how frequently secondary copies are made (the *accumulation window*) and how long they take to propagate to a given level of the hierarchy (the *propagation window*), thus determining how much data loss might be experienced after a disaster. (Table 2 provides examples of these parameters for our experimental environment.) Evaluation of the models also determines how the techniques consume resources, such as storage device and network link bandwidth. Section 3.2 describes how this framework is extended to model resource contention in multi-application environments.

2.2 Application workload characteristics

To estimate the bandwidth and capacity requirements for creating secondary copies, we must understand the application's data access patterns. Techniques that retain a full copy of the data require the solver to understand the *capacity* of the dataset. Techniques that immediately propagate updates, such as synchronous mirroring, require an understanding of the application's *peak (non-unique) update rate* to determine the required network bandwidth. Asynchronous mirroring techniques require network bandwidth to support the application's *average (non-unique) update rate*. Techniques that periodically create secondary copies require the solver to understand the *unique update rate*. Finally, recovery techniques that redirect application computation, such as failover, also require the solver to understand the application's *average access (read + write) rate*. Table 1 provides examples of these parameters for the workloads considered in our experiments.

2.3 Device infrastructure

Data protection and recovery techniques employ storage devices, such as disk arrays, tape libraries, and network interconnects, to store and propagate copies, respectively. Recovery techniques like failover also employ computational resources.

As in [13], we model several aspects of device resource configuration. Capacity and bandwidth are allocated in discrete units. Each device has *capacity* and *bandwidth constraints* that limit the number of applications and data protection techniques that can simultaneously use that device.

In addition, we model the *outlay costs* necessary to use the device infrastructure. Each device has a *fixed cost* associated with acquiring an instance of that device type (e.g., the cost of a disk array enclosure). A device may also have a *per-capacity cost* and a *per-bandwidth cost* (e.g., the costs of tape cartridges and tape drives for a tape library). The resource costs cover the direct and indirect costs of using the resources, including the hardware (e.g., purchase or lease price), software licenses, service contracts and management costs and facility costs. Table 3 provides examples of these cost, capacity and bandwidth parameters for the device types used in our experiments.

The solution must completely describe the employed resources, including each of the available sites, the different storage devices employed at each site, the interconnects between the sites, and their parameters.

2.4 Failure model

The primary copy of an application's dataset faces a variety of failures after deployment, including hardware failures, software failures, human errors and site and regional disasters. A failure scenario is described by its *failure scope*, or the set of failed storage and interconnect devices. Examples include *primary data object failure*, *primary disk array failure* and *primary site disaster*. A primary data object failure indicates the loss or corruption of the data due to human or software error without a corresponding hardware failure. Each failure scenario also has a *likelihood of occurrence*, which describes the expected likelihood of experiencing that failure.

Failed applications incur penalty costs due to the unavailability and loss of data. We model these penalties as described in [11]. In particular, a *data outage penalty rate* describes the cost (e.g., in US$ per hour) of data unavailability. After a failure, data is recovered from a secondary copy, which may be out-of-date relative to the time of the failure, thus implying the loss of recent updates. The *recent data loss penalty rate* describes the cost (e.g., in US$ per hour) of recent data loss. Table 1 provides examples for the workloads used in our experiments.

2.5 Solution cost

In order to choose among alternative designs, the design tool must assign a cost to each potential solution. The overall cost of the storage solution includes the outlays for the employed resources and the penalties for recovering the application data. Outlay costs are calculated for the entire resource infrastructure, including the fixed and incremental costs of the devices and the facilities costs of the data center sites.

The models are evaluated (as described in Section 3.2) to determine the recovery time (data outage time) and recent data loss time for each failure scenario. The computed recent data loss penalty and data outage penalty from each scenario is weighted by the failure's likelihood. The overall penalty cost is the sum of the weighted recent data loss and data outage penalties over all failure scenarios and all application workloads.

To provide a meaningful sum of the outlays and penalties, both cost categories must be calculated over a common time frame. Since most businesses look at annual costs, our models amortize the purchase price of devices over their expected lifetime (which is chosen to be three years). Similarly, the likelihood of failure is converted to an annual expected failure likelihood.

2.6 Putting it all together: problem statement

Given a description of application penalty rates, access characteristics, topology of data center sites, maximum number of permitted devices among all sites, and failure scenarios, our goal is to determine: 1) the combination of data protection and recovery techniques for each application; 2) the quantitative configuration parameters associated with each data protection technique; 3) the device resources needed to support normal and recovery operation; and 4) the mapping of primary and secondary data copies onto the provisioned resource instances, such that the overall cost of the solution, including both outlays and expected penalties, is minimized. The next section describes the approach we take to make these design choices.

3 Solution techniques

Our overall approach is to decompose the problem into two sets of decisions: (1) the choice of data protection techniques and the storage resources they use and (2) the choice of configuration parameter values for the high-level design decisions (e.g., the frequency of backups and the number of disks in the disk arrays). We chose to decompose the problem because the parameter space is too large to be explored efficiently in a single pass. Since different data protection techniques have different configuration parameters, selecting the data protection techniques first allows a more meaningful search for the configuration parameters and reduces the search space.

Figure 1 illustrates the architecture of the tool that embodies our general approach. It consists of a *design solver*, which selects data protection techniques for each application, and a *configuration solver*, which completes the design by selecting the configuration parameters for the chosen data protection techniques and the associated storage, network and computing resources. The user provides the applications' business requirements (expressed as penalty rates) and the applications' workload characteristics as inputs. The design tool uses this information to evaluate candidate storage designs and to produce a solution that attempts to minimize

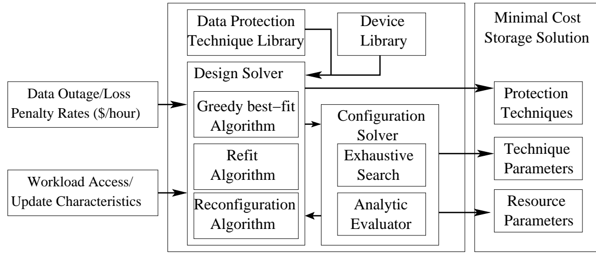

Figure 1. Automated design tool for dependable storage solutions

the overall cost. Many such complete designs are generated, and the design with the lowest cost is selected. The output of the design tool is a dependable storage design with near-optimal (minimal) cost. The next sections describe the operation of the design solver and configuration solver in more detail.

3.1 Design solver

The process of assigning data protection techniques and resources to application workloads can be thought of as a search on a graph of candidate partial designs. Each node in the graph is a design with some fraction (possibly all) of the application workloads assigned data protection techniques and corresponding resources. If there is an edge from node A to node B, then the design in node B can be obtained from the design in node A, either by adding an application workload (with the data protection technique and resource assignments) or by changing the data protection technique or resource assignments for one application workload.

The search consists of two stages. The first *greedy* stage starts with an empty node with no application workloads assigned and adds one application workload at a time until a feasible solution is found with all application workloads assigned. In the second, *refit* stage, the search explores the graph starting from this feasible initial node until it finds a local optimum. In both stages, each node along the way is evaluated by running the configuration solver to complete the design and computing the corresponding overall cost for the node's design. The search is repeated multiple times until a required computation time or until a specific criterion is satisfied. Since the steps in the search are randomized, all iterations of the search are expected to be different, thus enabling the search heuristic to escape the local minima. The best solution found over all the searches is returned. We describe the two stages of the search in more detail below.

3.1.1 Stage 1: Greedy best-fit algorithm

The greedy best-fit algorithm builds a storage solution by successively adding application workloads and their data protection techniques to the solution, assuming that the solution for the previously added application workloads remains constant. To add a new application, the algorithm exhaustively tries all possible data protection techniques for the chosen application and picks the one that minimizes the cost. The order in which the applications are added determines the quality of the solution. The algorithm chooses each application randomly, where the likelihood of choosing a particular application is based on the sum of its penalty rates. This approach favors applications with stringent requirements, and the probabilistic selection provides slightly different answers on successive iterations, allowing the algorithm to escape local minima. The greedy best-fit algorithm terminates when all application workloads are assigned data protection technique(s). The algorithm restarts if it determines that it is not feasible to add the remaining application workloads to the current layout. Lines 3 through 8 in Algorithm 1 present stage 1 of the design solver. The function *reconfiguration* is described in Section 3.1.3. The greedily chosen feasible design is passed on to the refit stage for further refinement.

3.1.2 Stage 2: Refit algorithm

Starting from the greedily chosen design, the refit stage iteratively searches its neighborhood in the design graph until a local optimum is found. In each iteration (Lines 14 through 42 in Algorithm 1), the algorithm randomly selects b (typically, 3) neighbors of the initial node and does a depth-first search up to a level d (typically, 5) from each neighbor (Lines 21 through 35 in Algorithm 1). At each level, b randomly selected neighbors are evaluated, and the best (minimal-cost) node is selected. At the end of the search, the best node found in that iteration is selected as the initial node for the next iteration. A local optimum is detected when the iteration completes without any improvement. Traversing an edge in the design graph in the refit stage requires a *reconfiguration*, in which the data protection techniques and resources assigned to an application workload are changed.

3.1.3 Reconfiguration algorithm

Reconfiguring an application is done by first removing the application from the design, and then providing it with a new data protection design and data layout. Although the choice of the application to reconfigure is random, the selection is biased towards applications that contribute the most towards the overall cost of the design, so that the reconfiguration has a higher chance of reducing the cost significantly. The algorithm chooses the data protection technique(s) to protect the application probabilistically, based on the application's requirements. To restrict the space of possible data protection configurations to explore, we divide both the applications and the data protection techniques into a small number of classes (e.g., three). Applications are categorized based on fixed thresholds of the sum of their penalty rates. Data protection techniques are categorized according to the level of protection they provide against downtime and data loss. In descending order of protection, categories include techniques using mirroring with failover

Algorithm 1 Design Solver

1: Let
- N = number of applications
- A_i = i^{th} application with its parameters, $1 \leq i \leq N$
- unC = unassigned set of applications, i.e.$\bigcup_{i=0}^{N} A_i$
- $curC$ = $\emptyset$, current partial candidate solution
- $newC$ = new partial candidate solution
- $bestC$ = minimum candidate solution seen so far
- d = level of depth of the search of a sibling tree
- b = breadth of search of sub-tree
- $stack[b * d]$ = stack
- tos = top of stack
- *rfgCnt* = reconfiguration iteration count

2: *rfgCnt* $= 0$
  {*STAGE 1: greedy best-fit algorithm*}
3: **repeat**
4:   choose A_i such that sum of recovery time and data loss penalty rate is maximum from the set of applications in unC.
5:   **reconfiguration**(curC,A_i)
6:   $newC$ =**configuration_solver**(curC)
7:   $curC += A_i, unC -= A_i$
8: **until** ($unC = \emptyset$)
  {*STAGE 2: re-fit algorithm*}
9: $tos = 0$
10: $bestC = curC$
11: **if** *rfgCnt* $>$ *threshold* **then**
12:   terminate solver, return $bestC$ to User.
13: **end if**
14: **repeat**
15:   $stack[tos++] = curC$
16:   **for** $i = 1$ **to** b **do**
17:     $curC$ = **reconfiguration**($curC$)
18:     $curC$ = **configuration_solver**($curC$)
19:     $stack[tos++] = curC$
20:     $j = 0$
21:     **while** ($j <= d$) **do**
22:       $popCnt = 0$
23:       **for** $k = 1$ **to** b **do**
24:         $newC$ = **reconfiguration**($curC$)
25:         $newC$ =**configuration_solver**($newC$)
26:         **if** ($cost(newC) < cost(curC)$) **then**
27:           $stack[tos++] = curC$
28:           $popCnt = popCnt + 1$
29:         **end if**
30:       **end for**
31:       $curC$ = **find_min**($stack$,$popCnt$)
      {find minimum cost solution for the current level}
32:       $tos = tos - popCnt$
33:       $stack[tos++] = curC$
34:       $j = j + 1$
35:     **end while**
36:     $curC = stack[0]$
    {restart search for the next sibling of the initial node}
37:   **end for**
38:   $bestC$ = **find_min**($stack$,tos)
39:   $tos = 0$
40:   $curC = stack[tos++] = bestC$
41:   *rfgCnt* = *rfgCnt* + 1
  {if sufficient progress check fails, go back to best-fit}
42: **until** (*rfgCnt* $> max$) || ($user\text{-}defined\ termination\ condition$)
43: **return** $bestC$

recovery, techniques using mirroring with data restoration and techniques using backup alone. For a given application class, the algorithm considers only data protection configurations from the corresponding class or better. It evaluates all such eligible configurations to determine their incremental costs in the context of the full candidate solution. The algorithm chooses one of the eligible techniques randomly, with a bias towards picking inexpensive techniques. More precisely, technique dpt is chosen with probability $1 - cost_dpt / \sum^{all_eligible_dpt} cost_dpt$.

The algorithm next determines the data layout (choices of devices and their layout on the sites) for the application. The resources that can be used are limited to those that can support the chosen data protection technique. Currently unused resources are excluded, unless the resource list is empty. The resources are selected randomly; the selection is biased towards under-utilized resources (to encourage load balancing) and against those that have been used for this application workload in previously explored configurations (to encourage diversity of choices). More precisely, the selection probability of each eligible resource A is proportional to $\alpha_{util} * (1 - util(A)) + (1 - \alpha_{util}) * (1 - usage(A))$, where $util(A)$ is the current utilization of A, $usage(A)$ is the fraction of times that A has previously been used for this application workload, and α_{util} is a weight between zero and one. We generally set α_{util} close to one, favoring load-balance over historical diversity. The new choices of data protection technique(s) and resource layout are added to the design solution and returned to the design solver (Lines 5, 17 and 24 in Algorithm 1).

3.2 Configuration solver

Given the partial candidate solution provided by the design solver, the configuration solver optimizes the configuration parameter values to obtain a complete candidate solution (Lines 6, 18 and 25 in Algorithm 1). It performs an exhaustive search over a discretized range of values for each of the parameters. These valid ranges of values are based on policies (e.g., the period between successive backups must be in 12-hour increments) and infrastructure deployment (e.g., a physical limit on the number of network links between two sites).

The configuration solver determines the recent data loss times and recovery times for each failed application under all failure scenarios. These times are used to compute the penalties for recovering the failed applications.

3.2.1 Recent data loss time

Upon failure of the primary copy, a secondary copy must be used to recover the data. The recent data loss time is the difference in time between the failure occurrence and the point in time represented by the secondary copy used for the recovery. The configuration solver applies the methodology described in [13] to determine how out-of-date each secondary copy is, and to choose which copy should be used

for recovery. The configuration parameter values determine an upper bound on the staleness of the most recent copy, based on how frequently copies were made and propagated. From these consistent secondary copies that are still accessible after the failure scenario, the solver chooses the copy that provides the minimum recent data loss.

3.2.2 Recovery time

Recovering an application from failure involves specific recovery tasks at each level of the recovery hierarchy. These tasks include repairing failed resources, copying consistent data back onto the primary disk arrays, reconfiguring the application, and more. Application and data protection workloads that are unaffected by the failure continue to run uninterrupted, using their assigned resources. The remaining bandwidth and capacity are made available for recovery operations. Scheduling recovery of failed applications is itself a complex problem; for simplicity, we assume the following precedents. If multiple recovery operations compete for the same resource, their execution is serialized according to a priority (the sum of each application's penalty rates). Recovery tasks for applications with higher penalty rates get higher priority, thus delaying the execution of lower-priority recovery tasks. The configuration solver simulates the recovery process to determine the recovery time for each failed application.

The configuration solver optimizes the resource-related parameters by first evaluating the recovery times for configurations containing the minimum resources required to support the applications and their data protection workloads. However, it is possible to shorten these initial computed recovery times, by adding resources to the system (e.g., additional network links or tape drives to provide more bandwidth). The algorithm continues to add resources until it no longer produces any cost savings. The configuration solver determines which set of configuration parameter values minimizes the overall cost and returns the fully specified candidate solution and its cost to the outer design solver.

4 Experimental results

We present experimental results to evaluate the design tool. In doing so, we compare the design produced by our design tool with that of a hypothetical human storage solution architect (approximated by a "human heuristic") and a random design selection algorithm. After we describe the heuristic, we compare our method with three types of results. We first describe a simple case study for a small environment, in order to build our intuition about the design tool's operation. We then study the scalability of our algorithms using a larger number of applications. Finally, we analyze the algorithm's sensitivity to failure likelihood.

4.1 Human heuristic

To understand the effectiveness of our design tool, we need a comparison point that approximates the behavior of a human storage solution architect. Based on our discussion with storage system architects, they categorize applications, data protection techniques and resources into different classes (gold, silver, bronze, etc.) based on their business requirements, features and capabilities. The architect applies the data protection techniques and resources from a given class to the applications in the corresponding class. Depending upon the availability of resources, the architect spreads the applications uniformly over the resource topology and sites to minimize the penalties due to failure.

Our "human heuristic" emulates this process by classifying the applications, data protection techniques and resources into three categories, as described in Section 3.1.3. The heuristic provides each application with data protection from the same or a better category of data protection technique. Each category might have multiple applications, so applications are assigned data protection techniques in a randomized priority order, based on the sum of the application's penalty rates. Similarly, there may be multiple data protection techniques in each class; the heuristic selects one of these techniques, where all of the techniques have the same probability of being selected. The set of required resources and sites is chosen such that applications are well-distributed over all the sites. Once all the applications have been assigned a data protection design, the heuristic uses the configuration solver to optimize the remaining configuration parameters.

The heuristic determines if the assignments make the storage protection solution infeasible; if so, it restarts the algorithm. After a fixed number of iterations, it returns without a solution. Since the choices are random in nature, the human heuristic is run for a bounded execution time, and the minimum-cost solution is selected.

4.2 Environment

Our experiments use a common set of input parameters for application business requirements and workload characteristics, data protection technique alternatives, and resource capabilities and costs. Table 1 describes the application classes used in our experiments. The penalty rate magnitudes are based on market research [6], and the application workload characteristics are based on scaled versions of the cello2002 workload described in [11]. Table 2 summarizes the data protection alternatives considered by our algorithms. Table 3 enumerates resource characteristics for disk arrays, tape libraries, network links and data center sites. The likelihoods of an application data object failure (e.g., due to user error or software malfunction), a disk array failure, and a data center site disaster are set to once in three years, once in three years, and once in five years, respectively.

Table 1. Application business requirements and workload characteristics

Type	Outage penalty rate ($/hr)	Recent loss penalty rate ($/hr)	Data size (GB)	Avg update rate (MB/sec)	Peak update rate (MB/sec)	Average access rate (MB/sec)	Category
Central banking (B): critical, expects zero data loss and data outage loss							
B	$5M	$5M	1300	5	50	50	Gold
Company web service (W): high transaction volume, modest recent data loss, zero outages							
W	$5M	$5K	4300	2	20	20	Silver
Consumer banking (C): high transaction volume, expects zero recent data loss, modest outages							
C	$5K	$5M	4300	1	10	10	Silver
Student accounts (S): student accounts, tolerant to data loss and vulnerability							
S	$5K	$5K	500	0.5	5	5	Bronze

Table 2. Data protection techniques

Data protection technique type	Reconstruct (R) or Failover (F)	Category	Level 1 snapshot (S) or mirror (M)			Level 2 tape library in days		Level 3 vault in days	
				accWin	propWin	accWin	propWin	accWin	propWin
Synchronous mirror with backup	Failover	Gold	M	0.5 min	n/w				
			S	12 hr	tape	7 days	tape	28 days	1 day
Synchronous mirror with backup	Reconstruct	Silver	M	0.5 min	n/w				
			S	12 hr	tape	7 days	tape	28 days	1 day
Asynchronous mirror with backup	Failover	Gold	M	10 min	n/w				
			S	12 hr	tape	7 days	tape	28 days	1 day
Asynchronous mirror with backup	Reconstruct	Silver	M	10 min	n/w				
			S	12 hr	tape	7 days	tape	28 days	1 day
Synchronous mirror	Failover	Gold	M	0.5 min	n/w				
Synchronous mirror	Reconstruct	Silver	M	0.5 min	n/w				
Asynchronous mirror	Failover	Gold	M	10 min	n/w				
Asynchronous mirror	Reconstruct	Silver	M	10 min	n/w				
Tape backup	Reconstruct	Bronze	S	12 hr	tape	7 days	tape	28 days	1 day
note:"n/w" and "tape" indicate that the propagation delay depends on the available network or tape library bandwidth									

Table 3. Resource description (unamortized purchase price)

Resource type	Class	Fixed cost ($)	Fixed BW (MB/s)	Incremental cost ($) per unit capacity	Incremental cost ($) per unit BW	Total number of capacity (units)	Total number of BW (units)	Capacity per unit (GB)	BW per unit (MB/s)
Disk array (XP1200)	High	375,000	512	8723		1024		143	25
Disk array (EVA800)	Med	123,000	256	3720		512		143	10
Disk array (MSA1500)	Low	123,000	128	3720		128		143	8
Tape Library	High	141,000	2400		18,400	720	24	60	120
Tape Library	Med	76,000	400		10,400	120	4	60	120
Network	High		640		500,000		32		20
Network	Med		160		200,00		16		10
Compute	High		125,000						
Site		1,000,000							

Table 4. Data protection solution chosen by design tool for peer sites

App	Type	Data protection technique	Primary site	Site $P1$ array	Site $P1$ tapelib	Site $P2$ array	Site $P2$ tapelib	network
1	B	Async mirror (F) with backup	P2	✓		✓	✓	✓
2	C	Sync mirror (R) with backup	P1	✓	✓	✓		✓
3	W	Async mirror (F) with backup	P1	✓	✓	✓		✓
4	S	Tape backup	P1	✓	✓			
5	B	Async mirror (F) with backup	P1	✓	✓	✓		✓
6	C	Sync mirror (R) with backup	P1	✓	✓	✓		✓
7	W	Sync mirror (F) with backup	P1	✓	✓	✓		✓
8	S	Tape backup	P2			✓	✓	

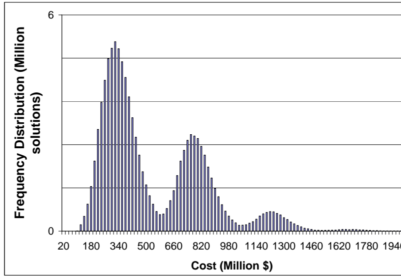

Figure 2. Distribution of data protection solution costs of peer sites

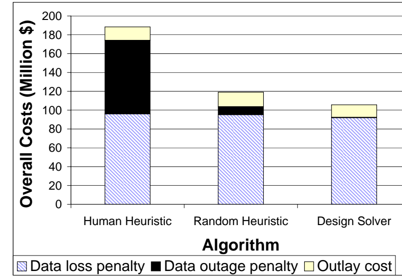

Figure 3. Comparison of algorithm data protection solution costs for peer sites

4.3 Simple case study: peer sites

To build our intuition about the solution space and the algorithms' behavior, we model a simple peer environment where a pair of sites serves as the primary site for a fraction of the applications and as a secondary site for the applications served primarily by its peer. This scenario models a multi-site corporation or service provider.

We want to deploy eight applications on two peer sites $P1$ and $P2$. Each site can accommodate a maximum of two disk arrays (e.g., one high-end and one low-end), a single tape library and compute resources for eight applications. A network with a capacity of up to 32 links connects the two sites. We execute each heuristic for a fixed time of thirty minutes.

4.3.1 Solution space insight

The parameter space of the dependable storage solution problem is extremely large. Even the partial configuration parameter space is about x^t, where $x = d^a$, d is the number of primary disk arrays, a is the number of applications deployed and t is the number of data protection techniques. As a result, it is intractable to obtain the optimal solution for comparison with our heuristics' solutions. Instead, we estimate solution quality by randomly sampling a large collection of solutions and evaluating their overall costs. Using this technique, we can estimate the quality of the heuristics' solutions in terms of where they reside in the empirical distribution of solutions.

Figure 2 illustrates the distribution of the peer sites' solution space, which is empirically determined from about one hundred million solutions. We observe that solution costs vary by more than an order of magnitude across the distribution. The goal of any heuristic is to pick solutions on the left side of the graph.

The distribution of solution costs is multi-modal, where each mode corresponds to a different set of choices being made for the design tradeoffs. Low-cost solutions protect applications with stringent requirements by increasing resource outlay expenditures to decrease penalties. Protection for applications with more relaxed requirements may be able to leverage the resources already in place for the more stringent applications. Higher-cost solutions provide inadequate protection for workloads with stringent requirements and thus incur high penalties.

4.3.2 Solution to case study

Table 4 describes the data protection solution chosen for each application by the automated design tool. As expected, applications with high data outage penalty rates always employ failover for recovery. It is cheaper to provide additional network links and compute resources to support failover than to incur penalties for recovery techniques that take longer. All applications employ some form of tape backup to support recovery from user errors and software malfunctions.

Counter to intuition, we note that the central banking applications (1 and 5) use asynchronous mirroring instead of synchronous mirroring. The increased recent data loss penalty for asynchronous mirroring is small, relative to the outlay for the additional resources to support synchronous mirroring. Therefore, the design tool chooses asynchronous mirroring over synchronous mirroring.

Figure 3 compares the cost of the outlays, data loss penalty and data outage penalty among the three different heuristics.

The design tool's solution costs roughly 1.9X less than the human heuristic solution and 1.3X less than the random heuristic's solution. The design tool's solutions fall within the lowest cost percentile of the solution cost space.

4.4 Algorithm scalability

Having built an intuition for the solution space, consider an environment with four sites, each with the potential to support two types of disk arrays, one tape library, compute resources and six network links that connect all the sites together. We assume the classes of applications described in Table 1 and the failure model used in Section 4.3. The environment is scaled by four applications at a time, one from each class. Each heuristic is run for thirty minutes for each experiment.

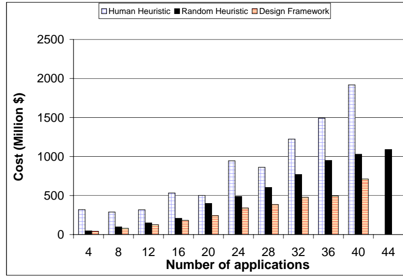

Figure 4. Design tool's scalability for the scenario with fully connected sites

Figure 4 compares the scalability of the three heuristics for the described environment. The design tool consistently provides better solutions than the random and human heuristics. These solutions are cheaper by a factor of 2X to 3X. The human heuristic fares poorly due to its inefficient layout strategy.

With the scaling of applications in a fixed-resource environment, determining even a feasible solution becomes a challenging hurdle. With severe resource restrictions, reconfiguration of a candidate solution takes a considerable number of retries to obtain the next partial candidate solution. The human heuristic and design solver fail to find a feasible solution for or more applications in this environment, due to the fixed resource constraints. The random heuristic succeeds at finding feasible solutions, even at this large scale, because it randomly generates data protection designs, which can be tested for feasibility fairly quickly.

4.5 Sensitivity to failure likelihood

Our final experiments explore the sensitivity of the design solver algorithm to failure likelihood. We vary the failure likelihoods for an environment with 16 applications and four fully connected sites. Data object failure frequency is varied from twice a year to once in ten years. Disk array failure frequency is varied from once in two years to once in twenty years, and site disaster frequency is varied from once in five years to once in fifty years. When they are not being varied, the frequencies of data object, disk and site failures are fixed at twice a year, once in five years and once in twenty years, respectively.

Figures 5, 6, and 7 plot the design tool's solution costs as a function of the likelihood of data object failures, disk array failures and site disasters, respectively. We observe that solution cost is relatively insensitive to the disk and site failure likelihood. The algorithm is able to compensate for the increased penalties from more frequent failures by slightly increasing its resource allocations (and subsequent outlay costs). After a certain threshold of the likelihood of data object failure, the solver is no longer able to compensate for the increased penalties by allocating additional resources. This failure sensitivity analysis lets the designer determine the range of failure likelihood for which the solution would

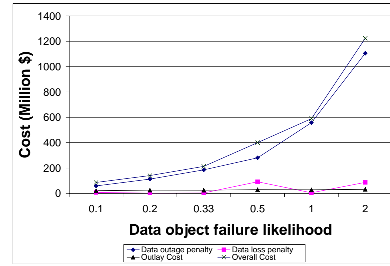

Figure 5. Design tool's sensitivity to the likelihood of data object failure

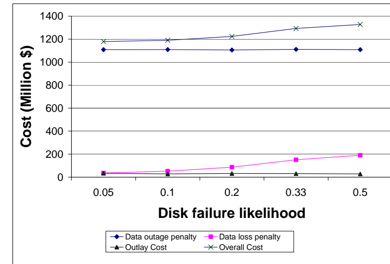

Figure 6. Design tool's sensitivity to the likelihood of disk failure

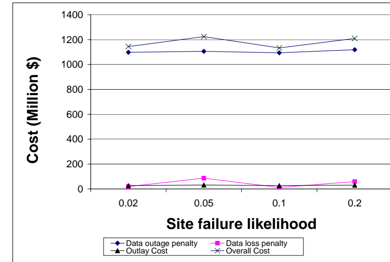

Figure 7. Design tool's sensitivity to the likelihood of site failure

protect the applications. Using this information, architects can design solutions suitable for the observed likelihood of failure.

5 Related Work

There is substantial work [1, 2, 3] in the design of storage systems to meet various performance and reliability goals at the lowest cost, but it only considers online reliability techniques such as RAID. Keeton et al., explore methods for dependable storage design in the context of a single application and a single dependability technique [11]; this paper considers multiple applications and combinations of techniques, which is a much more complex problem. In the area of modeling dependable storage system behavior, Keeton and Merchant present a framework for evaluating the recovery time and recent data loss for a single application protected by a combination of techniques [13]; more recent work by this group examines how to schedule recovery op-

erations for multiple workloads [12]. These papers consider only the dependability evaluation of an existing storage system, but do not consider how to design the system in the first place, which is the topic of this paper.

Direct search methods are some of the best known techniques for unconstrained optimization [14]. These techniques do not make any assumption about the underlying parameter space, but rather optimize depending upon the value of the objective function. These techniques are better suited to optimize continuous parameters. Pure combinatorial optimization problems are concerned with the efficient allocation of limited resources to meet the desired objectives [8]. These techniques expect prior knowledge of the bounds on the available resources to optimize. The upper bound on the available resources makes it computationally expensive to compute all possible combinations of allocations to solve the storage design problem using combinatorial optimization techniques such as linear or integer programming. In addition, the storage design problem for data protection requires the optimization of both continuous and discrete parameters, making it significantly harder. Local search heuristic techniques such as simulated annealing, tabu search [7] and local search are efficient in scenarios where the underlying structure of the parameter space is known [15]. Without sufficient information about the underlying structure, we perform better by exploring a much larger space at each local region, as demonstrated by our algorithm.

6 Conclusion

Designing a storage system to meet dependability goals in a multi-application environment is difficult. Interactions between the workloads, both in normal operational modes and recovery modes, lead to a large design space. Moreover, the design space lacks an inherent structure for traditional search techniques to exploit to determine an optimal solution.

This paper makes several contributions towards the goal of near-optimal dependable storage designs. Our search heuristic provides an intelligent method for exploring this unstructured design space. We decompose the problem into two stages, first determining which data protection techniques should be applied to each application, and then determining how to set the configuration parameters for these techniques and the resources they use. This decomposition reduces the size of the search space, allowing our algorithm to focus on the most relevant regions to achieve a near-optimal solution. In addition, we extend the abstractions for modeling single-application recovery from [13] to handle the interactions of multiple applications.

We compare the operation of our design tool's search heuristic with the ad hoc approaches used by human designers today. For the examples we consider, the automated design framework consistently generates solutions that are two to three times better than the solutions provided by the ad hoc approach. These improvements translate into savings of millions of dollars in the expected cost of deploying and recovering the resulting storage systems.

References

[1] G. A. Alvarez. Minerva: an automated resource provisioning tool for large-scale storage systems. *ACM Transactions on Computer Systems*, 19(4):483–518, Nov. 2001.

[2] E. Anderson et. al. Hippodrome: running circles around storage administrators. In *Proc. USENIX Conf. on File and Storage Technologies (FAST)*, pages 175–188, Jan. 2002.

[3] E. Anderson et. al. Selecting RAID levels for disk arrays. In *Proc. USENIX Conf. File and Storage Technologies (FAST)*, pages 189–201, Jan. 2002.

[4] A. Azagury, M. E. Factor, and J. Satran. Point-in-Time copy: Yesterday, today and tomorrow. In *Proc. IEEE/NASA Conf. Mass Storage Systems (MSS)*, pages 259–270, Apr. 2002.

[5] A. Chervenak, V. Vellanki, and Z. Kurmas. Protecting file systems: A survey of backup techniques. In *Proc. IEEE/NASA Conf. MSS*, pages 17 – 31, Mar. 1998.

[6] Eagle Rock Alliance Ltd. Online survey results: 2001 cost of downtime. http://contingencyplanningresearch.com/2001 Survey.pdf, Aug. 2001.

[7] F. Glover. Tabu search methods in artificial intelligence and operations research. *ORSA Artificial Intelligence Newsletters*, 1, 1987.

[8] M. Groetschel. Theoretical and practical aspects of combinatorial problem solving. In *Proc. 3rd ACM-SIAM Symp. on Discrete Algorithms (SODA)*, page 195, 1992.

[9] Hewlett-Packard Development Co. *HP OpenView Storage Data protector Administrator's Guide*, Oct. 2004. Mfg. Part Number B6960-90106, Release A.05.50.

[10] M. Ji, A. Veitch, and J. Wilkes. Seneca: remote mirroring done write. In *Proc. USENIX Technical Conf. (USENIX'03)*, pages 253–268, June 2003.

[11] K. Keeton et al. Designing for disasters. In *Proc. 3rd USENIX Conf. File and Storage Technologies (FAST)*, pages 59–72, Mar. 2004.

[12] K. Keeton et al. On the road to recovery: restoring data after disasters. In *Proc. European Systems Conf. (EuroSys)*, April 2006.

[13] K. Keeton and A. Merchant. A framework for evaluating storage system dependability. In *Proc. 2004 Intl. Conf. on Dependable Systems and Networks (DSN)*, pages 877–886, June 2004.

[14] T. G. Kolda, R. M. Lewis, and V. Torczon. Optimization by direct search: New perspectives on some classical and modern methods. In *Society for Industrial and Applied Mathematics (SIAM) Review*, volume 45, pages 385 – 482, july 2003.

[15] C. H. Papadimitriou and K. Steiglitz. *Combinatorial optimization : algorithms and complexity*. Dover Publications, 1998.

[16] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proc. ACM SIGMOD Conf.*, pages 109–116, June 1988.

[17] R. Schulman. *Disaster Recovery Issues and Solutions*. Hitachi Data Systems White paper, Sept. 2004.

[18] W.-D. Zhu et al. *IBM Content Manager Backup/Recovery and High Availability: Strategies, Options and Procedures*. IBM Redbook, Mar. 2004.