Designing Dependable Storage Solutions for Shared Application Environments

Shravan Gaonkar, *Student Member, IEEE,* Kimberly Keeton, *Member, IEEE,*
Arif Merchant, *Member, IEEE,* and William H. Sanders, *Fellow, IEEE*

***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 architects 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 intelligent exploration of 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 produces designs that cost up to two times less in initial outlays and expected data penalties than the designs produced by an emulated human design process. Additionally, we compare our design tool to a random search heuristic and a genetic algorithm meta-heuristic, and show that our approach consistently produces better designs for the cases we have studied. Finally, we study the sensitivity of our design tool to several input parameters.**

***Index Terms*—data protection techniques, design space exploration, discrete event simulation, genetic algorithm, search heuristic, storage system design**

I. Introduction

BUSINESSES today rely on their IT infrastructures, and events that cause data unavailability or loss can have expensive, or even catastrophic, consequences. Such events can include natural disasters, hardware 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. These techniques have limitations. 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 architect 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 policy needs to specify the frequency of the backups and whether the backups will be full or incremental. The architect must verify that the design will meet normal operational performance requirements (for example, the backups will complete overnight), and also 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, architects 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 methods to automatically design a dependable storage system that uses a single technique to protect a single application workload [1], and algorithms to evaluate the recovery behavior of a single application workload protected by a combination of techniques [2]. Storage system design for multi-application environments presents even greater challenges. First, the design space of data protection techniques 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, relative to what it would be if each application operated in isolation. Third, designing for multiple applications may prompt design decisions that are different from those that would result 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 workloads and a less expensive array for less important workloads.

Our contributions include search heuristics for intelligent exploration of the large design space, as well as modeling techniques for capturing interactions between applications during recovery. These techniques extend earlier single-

Manuscript received March, 2007; revised December, 2007; and accepted June, 2008.

application models [2] 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 and to those produced by a random search heuristic and a genetic algorithm meta-heuristic. For the scenarios we study, we find that our approach's solutions reduce overall system costs due to equipment and software outlays and data outage and loss penalties by at least a factor of two when compared to the human design choices. Furthermore, our approach consistently produces better solutions than the random heuristic and genetic algorithm. Finally, we study the sensitivity of our approach's solutions to several parameters, including the number of applications to be protected, the bandwidth and capacity characteristics of those applications, the likelihood of failures, and algorithm execution time.

The remainder of this paper is organized as follows. Section II formulates the problem of designing storage systems to protect application data. In Section III, we describe our design methods. In Section IV, we evaluate our approach quantitatively. Section V presents related work, and Section VI concludes the paper.

II. 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, 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.

A. Data protection and recovery techniques

In order to protect applications against data loss and unavailability, it is necessary to make 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 [3] 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 [4], [5] or remote vaulting) provides resilience against site and regional disasters. Point-in-time [6] and backup [7], [8], [9] copies address application data object errors, like accidental deletion and software failures due to buggy software or virus infection, by permitting restoration of a previously consistent copy. Those 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. For the restoration case, data is copied from the secondary copy to the target site. For failover, the computation is simply transferred to the secondary mirror, without any data copy operations. 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 [2] 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. For example, a hierarchy might include the primary copy, intra-array snapshot, tape backup, and remote vaulting. Secondary copies made at one level of the hierarchy are periodically transferred to the next level of the hierarchy. For example, tape backups are periodically shipped offsite to the remote vault. For each level of the hierarchy, data protection technique parameters specify how frequently secondary copies are made (the *accumulation window*), how long they take to propagate to a given level of the hierarchy (the *propagation window*), and how long they are retained at that level (the *retention window*), thus determining how much data loss might be experienced after a disaster. (Table III 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 III-B describes how this framework is extended to model resource contention in multi-application environments.

B. Application workload characteristics

To estimate the bandwidth and capacity requirements for creating secondary copies, we must understand the application's data access patterns. Applications share common resources to perform backup of data. 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*. For a given period of time, the unique update rate measures the last update to a given location, omitting overwrites; it tells how much new data is generated between the creations of subsequent secondary copies. 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 II provides examples of these parameters for the workloads considered in our experiments.

C. Device infrastructure

Data protection and recovery techniques employ storage devices, such as disk arrays, tape libraries, and network interconnects, to store and propagate copies. Recovery techniques like failover also employ computational resources. As in [2], we model several aspects of device resource configuration. Each device has *capacity* and *bandwidth constraints* that limit the number of applications and data protection techniques that can simultaneously use that device. Capacity and bandwidth are allocated in discrete units, and we assume a linear additive model for resource consumption. 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, management costs, and facility costs. The solution must completely describe the employed resources, including each of the available sites, the different storage and computational devices employed at each site, the interconnects between the sites, and their parameters.

D. 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 annual likelihood of experiencing that failure.

We assume that primary disk array failures and primary site disasters are detected immediately, and that the desired point of recovery is the most recent point in time. For primary data object failures, we assume that there is a delay between the failure and the discovery of the failure(e.g., due to user error). The desired point of recovery is the time (in the past) of the failure. For any failure, we assume that the recent data loss is the failure detection delay (i.e., the updates made after the failure), plus any additional updates lost due to recovery from a point-in-time copy that is out of date, relative to the desired recovery point. For instance, the failure may have occurred just before a backup, resulting in the loss of all updates since the previous backup.

Failed applications incur penalty costs due to the unavailability and loss of data. We model these penalties as described in [1]. 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.

E. 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 design tool evaluates the models (as described in Section III-B) to determine the recovery time (data outage time) and recent data loss time for each failure scenario. It weights the computed data outage penalty and recent data loss penalty from each scenario by the likelihood of that failure. The overall penalty cost is the sum of the weighted data outage and recent data loss 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.

F. 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 making those design choices.

III. Solution techniques

Our overall approach is to decompose the problem into two sets of decisions: 1) *qualitative parameter decisions*, which relate to the choice of data protection techniques and the data layout over the resources (e.g., the choice of primary array, network link, or tape library) and 2) *quantitative parameter decisions*, which relate to 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

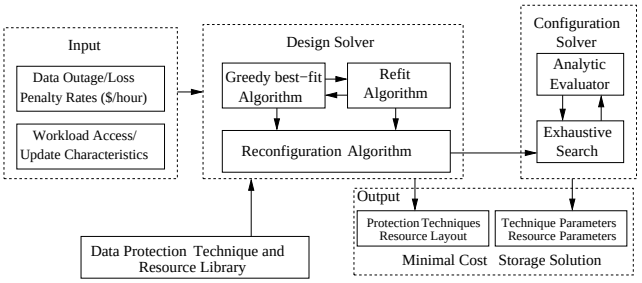

Fig. 1. Automated design tool for dependable storage solutions

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 quantitative 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 that information to evaluate candidate storage designs and to produce a solution that attempts to minimize 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.

A. 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 to which data protection techniques and corresponding resources have been assigned. 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 corresponding 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. First, the *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 the feasible initial node until it finds a local optimum. In both stages, the design solver evaluates each node by running the configuration solver to complete the design and compute 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 local minima. The best solution found over all the searches is returned. We describe the two stages of the search in more detail below.

1) Stage 1: Greedy best-fit algorithm: The greedy best-fit algorithm, shown in lines 3 through 8 of Algorithm 1, builds an initial 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 a factor that weights the sum of its penalty rates and the prior overall cost of the solution for this particular application. More specifically, application a is chosen with probability $p_p * 0.5 + p_c * 0.5$. Here, p_p is $\frac{\sum Pa}{\sum_{u \in U} Pu}$ and p_c is $\frac{\sum Ca}{\sum_{u \in U} Cu}$, where U is the set of unassigned applications, Px is the penalty rates defined on application x, and Cx is the cost of the solution for application x. We use a probabilistic variant because the greedy best-fit algorithm may be executed multiple times, and we want variation in the result in order to provide different starting points for the stage 2 algorithm. The approach of probabilistically selecting applications with high penalty rates earlier 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 techniques. The algorithm restarts if it determines that it is infeasible to add the remaining application workloads into the current solution. The function *reconfiguration* is described in Section III-A3. The greedily chosen feasible design is passed on to the refit stage for further refinement.

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) 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 probabilistic, 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

Algorithm 1 Design Solver

```
1: Let
     N          = number of applications
     A_i        = i^th application with its parameters, 1 ≤ i ≤ N
     unC        = unassigned set of applications, i.e., ⋃_{i=0}^{N} A_i
     curC       = ∅, 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 = ∅)
   {STAGE 2: refit 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-defined termination condition)
43: return bestC
```

of reducing the cost significantly. The algorithm first chooses the data protection technique(s) to protect the application, based on the application's requirements. 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.

The reconfiguration algorithm keeps track of the quality of designs. It tracks the design choices made for the qualitative parameters and correlates these choices with the cost of the design solution. Using the collected information on design decisions, the algorithm dynamically builds a matrix that correlates the quality of the design solution to qualitative parameter values. We call this matrix the *likelihood-correlation matrix* (LCM). Each time the reconfiguration algorithm changes the value of one of the input design decision variables, it uses the LCM to choose the new value. For example, consider a scenario with four applications and three data protection technique choices, as shown in Table I. Each row in Table I represents the likelihood of choosing a particular data protection technique for the application that would minimize the overall cost of the design. For instance, if application "Central banking" is being reconfigured, the algorithm is more likely to find a minimum-cost design solution if it chooses synchronous mirroring to protect the application. The reconfiguration algorithm maintains separate LCMs for each qualitative input decision variable, including the choice of data protection technique and the choice of data layout.

TABLE I
LIKELIHOOD-CORRELATION MATRIX

Application	DPT		
	Tape backup	Asynchronous mirroring	Synchronous mirroring
Student accounts	0.3	0.5	0.2
Consumer banking	0.5	0.3	0.2
Company Web service	0.3	0.1	0.6
Central banking	0.7	0.3	0.1

The entries in these LCMs are generated by observing the past history of design configurations that have already been explored. The reconfiguration algorithm maintains a history of the past N designs. Each time a design configuration is evaluated, it is added into this list, provided that its cost is less than twice the cost of the overall minimum-cost design solution explored by the search heuristics. If the design configuration's cost does not satisfy that condition, it is added into the list with a very low probability ($p < 0.01$). Both policies prevent the solver from getting stuck at the local minima. Once the list grows to size N, the oldest configuration is removed, provided that it is not the minimum cost design solution. Each entry in the LCM is computed as the ratio of the number of times a particular value was chosen to generate the design solutions maintained in the list to the total number of design solutions in the list. Looking back at the application "Central banking," if tape backup was used *X1* times, asynchronous mirror was used *X2* times, and synchronous mirror was used *X3* times, and these choices resulted in good designs, then entries in the "Central banking" row would be $\frac{X1}{N}$, $\frac{X2}{N}$, and $\frac{X3}{N}$, respectively.

To further 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 threshold values 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 recovery, techniques using mirroring

with data reconstruction, 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 proportional to $1 - cost\_dpt / \sum^{all\_eligible\_dpt} cost\_dpt$.

A new, unused resource is picked from the pool of resources only if all the currently used resources cannot accommodate the applications and their data protection techniques. This policy prevents the algorithm from having a large amount of underutilized resources in the final design solution. The resources are selected randomly; the selection is biased towards underutilized resources (to encourage load balancing) and against resources 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 $\alpha_{util} * (1 - util(A)) + (1 - \alpha_{util}) * (1 - LCM(A))$, where $util(A)$ is the current utilization of A, $LCM(A)$ is the fraction of times that A has previously been used for this application workload and resulted in a low-cost design solution, and α_{util} is a weight between zero and one. We generally set α_{util} greater than 0.5, 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).

B. Configuration solver (CS)

Given the partial candidate solution provided by the design solver, the configuration solver optimizes the quantitative 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. The 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. The times are used to compute the penalty costs for recovering the failed applications.

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 [2] to determine how out-of-date each secondary copy is, and to choose which copy should be used for recovery. The solver determines which secondary copies are still accessible after the failure scenario, and chooses the copy that provides the minimum recent data loss. Recent data loss is determined based on how frequently the secondary copies were made and propagated through the levels of the data protection hierarchy. More specifically, the recent data loss at level j is $\sum_{i=1}^{j} propWin_i + accWin_i$, where $propWin_i$ is the propagation window and $accWin_i$ is the accumulation window at level i.

2) Recovery time: Recovering an application from failure involves specific recovery tasks at each level of the recovery hierarchy, including repairing failed resources, copying consistent data back onto the primary disk arrays, and reconfiguring the application. The CS simulates the recovery process to determine the recovery time for each failed application. The CS builds a resource usage and scheduling chart (RC) for each available resource. Each row in this chart is a resource such as a disk array, tape backup, or network. The RC also stores information about the maximum capacity of the resource. Each entry in the row is a linked list that represents the utilization of the resource over time. Each entry has three records: the current time, the availability as a percentage, and the pointer to the next entry for the same resource. The CS reads this information entry by entry to determine the availability of a resource over time. The CS tries to minimize recovery time by maximizing the usage of available resources. Furthermore, the recovering applications are provided with exclusive access to available resources to eliminate contention. Algorithm 2 describes the process of determining the recovery time of the failed application.

Algorithm 2 Recovery time simulation using discrete event simulation

1: Let
   - r represent a resource
   - a represent an application
   - M represent the total number of resources
   - RC is a data structure that manages the resource utilization as applications are scheduled for recovery

2: Trigger failure based on the failure model (application failure, disk array failure, or site failure)
3: Mark failed resources in *RC* as unavailable
4: Initialize non-failed resources as 100% available

5: **for** each $r \in$ failed-resource-list **do**
6: Determine time t_r required to repair resource and bring it online
7: Add an entry $\{t_r$, 100, null$\}$ into the RC at location *r*
{This entry states that resource *r* is 100% availablet_r seconds after the failure}
8: **end for**

9: Add entries into RC to account for resources used for uninterrupted operation of applications and workloads that are unaffected by failure

10: **for** each *a* ∈ failed-application-list **do**
11: Determine resources *r* on which *a* depends
12: Determine order in which resources are required to restart application *a*
13: Update entry of *r* in *RC* to reflect resource usage by application *a*
14: Compute the time, t_a, when application *a* resumes operation based on available resources and application characteristics
15: Remove *a* from the failed-application-list
16: **end for**
{The order in which the application is chosen determines the cost}

17: Return values of t_a for all failed-applications

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, as indicated in line 9

of the algorithm. Scheduling recovery of failed applications is itself a complex problem; for simplicity, we assume the following policies. 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. If the sum of penalty rates cannot break the tie, the data loss penalty rates are used to prioritize the order of recovery.

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 techniques. 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). Adding resources may decrease the overall cost (because the decrease in recovery time penalties outweighs the increase in outlay costs), or increase the overall cost (because the increase in outlay costs for the additional resources does not provide sufficient recovery time savings). 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.

IV. Experimental results

We present experimental results to evaluate the design tool. In doing so, we compare the design produced by our design tool with those of a hypothetical human storage solution architect (approximated by a "human heuristic"), a random design-selection algorithm, and a genetic algorithm (a general meta-heuristic). After we describe the heuristics, 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 algorithm execution time, failure likelihood, application bandwidth, and capacity requirements.

A. 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. Our discussions with storage system architects revealed that they typically categorize applications, data protection techniques, and resources into different classes (e.g., gold, silver, and bronze) 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.

The "human heuristic" emulates this process by classifying the applications, data protection techniques, and resources into three categories. 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 each 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 eligible techniques have the same probability of being selected. Any technique whose class is the same or better than that of the application's class is an eligible technique for the application. 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 they do, it restarts the algorithm. After a fixed number of iterations, if no feasible solutions are found, it returns without a solution. Otherwise, since the choices are random in nature, the human heuristic is run for a bounded execution time, and the minimum-cost solution is selected.

B. Random search

One question that often arises is the density of the optimal solution in the design space. If a design problem has a large number of nearly optimal solutions, then a simple strategy of random exploration should suffice. Since it is usually not possible to explore the entire design space, we implement a random search to compare the quality of the solution obtained using our design solver to a random strategy that uses the same amount of computational resources. In the random strategy, each application is provided with a data protection technique and a layout on the resources using a uniform distribution. The random search strategy uses the configuration solver to optimize the quantitative parameters.

C. Genetic algorithm

Past research has explored Storage Area Network (SAN) design using genetic algorithms (GAs) to produce good designs [10]. We develop a similar generic genetic algorithm formulation to serve as a comparison point for the design solver.

1) Background: A GA is a computer simulation of biological evolution. The values of all the decision variables are represented as a DNA string (also called an *individual*), where each position in the string has a finite set of values. The GA keeps track of the fitness of the DNA string (individual) using a function (e.g., a simulation or analytical evaluation) that takes the string as input and returns a scalar value. A group of individuals together forms a population. The transition from one generation to the next is performed by crossover (mating) between two individuals, genetic mutation within a single individual, and selection of the fittest individuals for

the next generation. This process is repeated over successive generations of the population.

2) Genome encoding: To have a fair comparison with other algorithms, our GA encodes and expresses only the qualitative decision variables. To optimize the quantitative decision variables, the GA uses an exhaustive search to determine the optimal values. The genome string of a design configuration contains information about the choice of data protection technique and resources used by each application deployed. All the information on qualitative parameter values is encoded into a single sequence ordered on the basis of the application to which it belongs, as shown in Figure 2.

3) Algorithm: Algorithm 3 describes the details of our implementation. The initial individuals are generated randomly to form the initial population. The maximum population size is a user-tunable parameter currently set to a value of 10,000. The algorithm is run for several generations until a terminating criterion is reached. This criterion could be a limit on execution time or a limit on the number of successive generations, or a situation in which the solution has not improved for a defined amount of time. At each successive generation, the fitness of each individual is computed. Here the fitness function is the total cost of the design solution. A lower cost means a more fit individual. The unfit individuals are discarded stochastically (Line 5 in Algorithm 3) to keep diversity in the population and prevent the algorithm from getting stuck at local minima.

Algorithm 3 Genetic algorithm to design dependable storage

1: Let
   N be total number of individuals in the population
   f_i Fitness of an individual i, where f_i is the total cost of the design
2: **while** $(time_{running} < time_{allotted})$ **do**
3: Compute fitness f_i for each $i \in N$
4: Sort the individuals in a nondecreasing order based on f_i $\{f_{min}$ is the fittest individual$\}$
5: For each i, $N - n \leq i < N$, delete i with probability p, where $(M < \frac{n}{2})$
6: Generate $\frac{n}{2}$ new individuals using *mutation*
7: Generate $\frac{n}{2}$ new individuals using *crossover*
8: **end while**
9: Sort the individuals in a nondecreasing order based on f_i
10: Return the individual(s) with value equal to f_{min}

The next two steps in the algorithm are used to generate the individuals for the next generation of the population (Lines 6 and 7 in Algorithm 3). *Mutation* is a process by which the algorithm creates a new individual from a parent by choosing a random number of applications in the genome string and modifying all the genes of these chosen applications. We pick a random number (user-defined probability) of applications and replace all the genes (qualitative and associated quantitative parameter values) during the mutation process. *Crossover* is a process by which the algorithm creates a new individual (offspring) by recombining substrings of the encoded genome strings from each individual parent. We pick a random number (user-defined probability) of applications and cross-over (exchange parameter values for) all the genes for those applications. That enables the algorithm to modify the characteristics of an individual (the qualitative parameters) on a per-application basis.

D. 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 II describes the application classes used in our experiments. The penalty rate magnitudes are based on market research [11]. Table III summarizes the data protection alternatives considered by our algorithms. Table IV 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 five years, and once in ten years, respectively. For an application data object failure, we model the delay from when the failure occurs to when the failure is discovered as ten hours.

All our experiments were carried out on a 256-node Linux cluster running on the 2.6.9 kernel. Each node on the cluster is an AMD dual-processor machine with 1 to 8GB of RAM connected through an InfiniBand network to the cluster. The experiments were submitted as jobs to the cluster, where each job used a single processor core. All experiments were executed for thirty minutes, and the experiments were repeated thirty times. Each experimental result is the average of the thirty runs, and the error bar represents a 95% confidence interval.

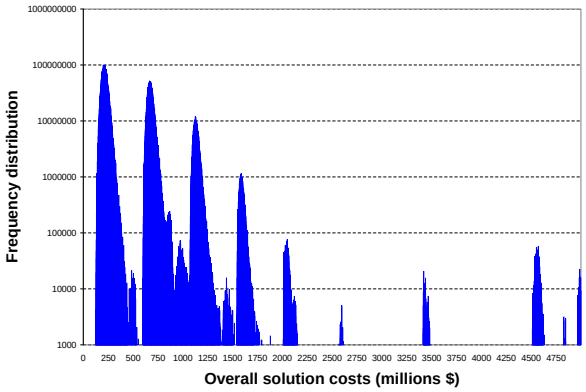

Fig. 3. Distribution of data protection solution costs of peer sites. Note that the Y axis is in log scale.

E. Simple case study: Peer sites

To build our intuition about the solution space and the algorithms' behavior, we modeled a simple peer environment in which a pair of sites serves as the primary site for a fraction of the applications and as a secondary site for the remaining fraction of the applications. 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.

Central banking							Consumer banking							Student account						
DPT	ST	P	S	T	V	N	DPT	ST	P	S	T	V	N	DPT	ST	P	S	T	V	N
9	0	1	-	5	-	-	1	1	2	4	-	6	7	1	2	1	1	-	-	6
Note: DPT = Data protection technique, ST = Site, P = Primary array, S = Secondary array, T = Tape library, V = Tape vault, N = Network																				

Fig. 2. Possible genome encoding of a design solution to deploy three applications. Values refer to the indexes of information presented in Table III and Table IV.

TABLE II
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	22	50	Gold
Company web service (W): high transaction volume, modest recent data loss, zero outages							
W	$5M	$5K	4300	2	10	20	Silver
Consumer banking (C): high transaction volume, expects zero recent data loss, modest outages							
C	$5K	$5M	4300	1	5	10	Silver
Student accounts (S): student accounts, tolerant to data loss and vulnerability							
S	$5K	$5K	500	0.5	2	5	Bronze

TABLE III
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	
				On	accWin	propWin	accWin	propWin	accWin	propWin
1	Split mirror with backup	Failover	Gold	M S	0.5 min 12 hr	n/w tape	7 days	tape	28 days	1 day
2	Split mirror with backup	Reconstruct	Silver	M S	0.5 min 12 hr	n/w tape	7 days	tape	28 days	1 day
3	Asynchronous mirror with backup	Failover	Gold	M S	10 min 12 hr	n/w tape	7 days	tape	28 days	1 day
4	Asynchronous mirror with backup	Reconstruct	Silver	M S	10 min 12 hr	n/w tape	7 days	tape	28 days	1 day
5	Split mirror	Failover	Gold	M	0.5 min	n/w				
6	Split mirror	Reconstruct	Silver	M	0.5 min	n/w				
7	Asynchronous mirror	Failover	Gold	M	10 min	n/w				
8	Asynchronous mirror	Reconstruct	Silver	M	10 min	n/w				
9	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 IV
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 amount of capacity (units)	Total amount of BW (units)	Capacity per unit (GB)	BW per unit (MB/s)
1	Disk array (XP1024)	High	375,000	512	8723		1024		143	25
2	Disk array (EVA8000)	Med	123,000	256	3720		512		143	10
3	Disk array (MSA1500)	Low	123,000	128	3720		128		143	8
4	Tape Library	High	141,000	2400		18,400	720	24	60	120
5	Tape Library	Med	76,000	400		10,400	120	4	60	120
6	Network	High		640		500,000		32		20
7	Network	Med		160		200,000		16		10
8	Compute	High		125,000						
9	Site		1,000,000							

TABLE V
DATA PROTECTION SOLUTIONS CHOSEN BY DESIGN TOOL FOR PEER SITES

App	Type	Data protection technique	Primary site	Site $P1$		Site $P2$		network
				array	tapelib	array	tapelib	
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			√	√	

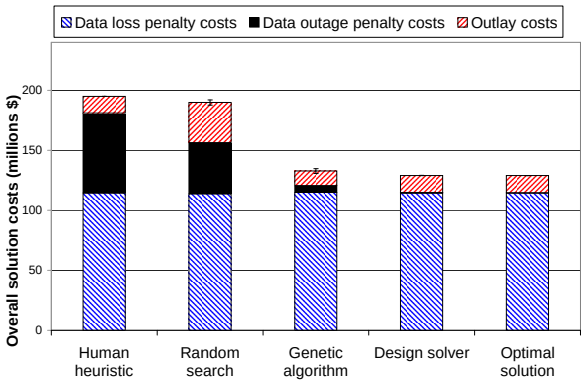

Fig. 4. Comparison among costs of search heuristic solutions and optimal solution for peer sites running eight applications

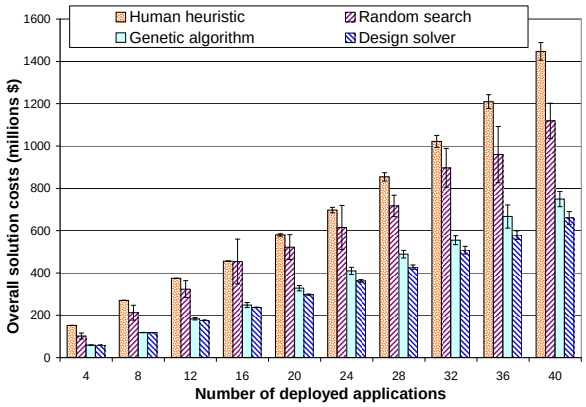

Fig. 5. Comparison among search heuristic algorithms as applications are scaled for a scenario with fully connected sites

1) Solution space insight: The parameter space of the dependable storage solution problem is extremely large. Even the partial qualitative parameter space is about $d^a * a^t$, where d is the number of primary disk arrays, a is the number of applications deployed, and t is the number of data protection techniques. Figure 3 illustrates the distribution of the peer sites' solution space, which is determined by exhaustive exploration of the design space. The size of the design space, considering only the qualitative variables, is about 43.1 billion alternative configurations (d = 2, a = 8, t = 9). Exploring the entire space takes about 280 computer hours. We observe that solution costs vary by more than an order of magnitude across the distribution. The goal of any heuristic is to quickly identify solutions on the left side of the graph.

The distribution of solution costs is multimodal, where each mode corresponds to a different set of choices being made for the design trade-offs. 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.

2) Solution to case study: Table V 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 4 compares the outlay, data loss penalty, and data outage penalty costs of the four different heuristics against the cost of the optimal solution.

For our case study, the design tool's solution costs roughly 1.5 times less than either the human heuristic's solution or the random heuristic's solution. Here, the design tool's search heuristic found one of the optimal solutions for the case study. Due to the simplicity of the case study, the solutions identified by the design solver and the genetic algorithm have similar costs. In the next section, we will see the impact of increasing the number of decision variables on the quality of the design solution.

F. Algorithm scalability

Having developed an intuitive understanding of the solution space, in this section we now examine the quality of solutions found by each of the algorithms as we scale the number of applications in a larger environment. The environment contains 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 that we are working with the classes of applications described in Table II and the failure model used in Section IV-D. We scale the environment by adding four applications at a time, one from each class.

Figure 5 compares the scalability of the four algorithms for the described environment. The design tool consistently provides better solutions than the other heuristics. More specifically, the design solver's solutions are 1.9 times to 2.6 times cheaper than those chosen by the human heuristic, 1.7 times to 1.9 times cheaper than those chosen by the random search

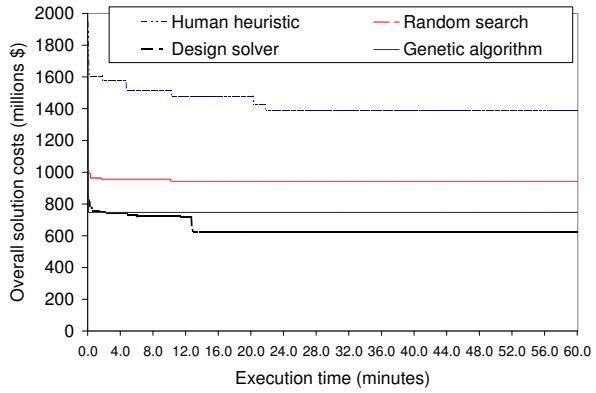

Fig. 6. Comparison of different heuristics' execution time sensitivity for a scenario with fully connected sites that have 40 applications

Fig. 7. Design solver's solution sensitivity to likelihood of data object failure

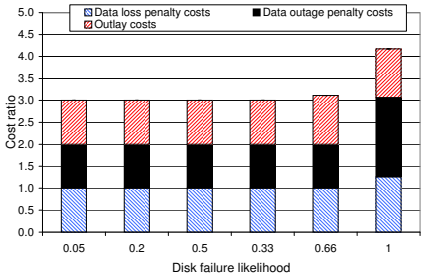

Fig. 8. Design solver's solution sensitivity to likelihood of disk failure

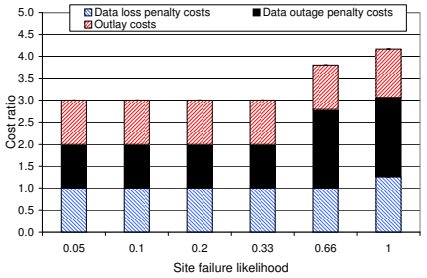

Fig. 9. Design solver's solution sensitivity to likelihood of site failure

heuristic, and up to 1.2 times cheaper than those chosen by the genetic algorithm. While the human heuristic fares poorly due to its inefficient layout strategy, the solutions provided by the random search heuristic and the genetic algorithm deteriorate as the solution space becomes larger. The key difference between the design solver search strategy and other approaches is its ability to determine the sensitivity of the final solution to the decision variable values, and to make decisions for the variables that have a larger impact on the design solution. In particular, the GA chooses the decision variables by looking at the overall cost of the system. As the number of design decision variables increases, this single overall cost metric masks the intricate relationships among the values of the variables. The design solver tracks these relationships, thus enabling it to determine good decision variable values to obtain near-optimal designs more quickly. Thus, the cost gap between the solution chosen by the design solver and the solutions chosen by the other heuristics tends to increase as the number of decision variables increases.

G. Sensitivity to execution time

Figure 6 illustrates the sensitivity of the four heuristics' solution quality to execution time. In our experiment, 40 applications were deployed on 4 fully connected data center sites. The cost of the initial solution obtained by each heuristic ranged from 800 million dollars to 1.9 billion dollars. The figure traces the cost of the minimal overall cost design solution for each of the heuristics by sampling the cost every 4 seconds until the terminating time of twelve hours. As we observe no further improvements after 30 minutes, we focus on the improvement gained in the first hour.

Although we traced the improvement in solution quality as a function of algorithm execution time for a large number of experiments (e.g., 30 replications for each algorithm for three different combinations of applications, or a total of 360 experiments), we present detailed results only for a single experiment for each heuristic algorithm as other experiments showed similar trends. The random search heuristic is very simple in nature, and the solutions it determines depend largely on chance. Given sufficient time, the random search will determine the optimal solution, while the human heuristic might never find the optimal solution because of the limited rules of thumb it employs in decision-making to explore large design spaces. The genetic algorithm and design solver are able to quickly reduce the costs in a few seconds using their specific strategies. As the design problem is scaled to larger sizes, the GA takes significantly longer to converge than the design solver does because the GA does not capture relationships among input decision variables. Its evolutionary strategy is completely independent of the decision variables and depends only on the objective function value. Unlike the GA, the design solver attempts to build a distribution of the decision variables that result in good solutions, which allows our approach to converge to better solutions.

H. Sensitivity to failure likelihood

In this section, we explore the sensitivity of the design solver's solutions to failure likelihood. These experiments were conducted using the environment from the simple case study in Section IV-E and the base application characteristics described in Table II. We varied the likelihood of all failure scopes from once in twenty years to once per year. When they were not being varied, the frequencies of data object, disk, and site failures were fixed at once in three years, once in five years, and once in ten years, respectively, as described in Section IV-D.

Figures 7, 8, and 9 plot the design tool's solution cost ratio as a function of the likelihood of data object failures, disk array failures, and site disasters, respectively. For all figures, the cost ratio is calculated relative to the default failure likelihood (0.33 for data objects, 0.2 for disk failures, and 0.1 for site failures). The columns are stacked to make it easy to view the relative change in overall costs. We observe that increasing the failure likelihood of disk or site failures increases the data outage penalty. The disk or site failures increase the recovery time because of the resource contention among the multiple recovering applications.

That failure sensitivity analysis lets a human storage architect determine the range of failure likelihoods for which the design solver's solution would adequately protect the applications. In this case study, the threshold is between 0.65–0.75 for data object, disk, and site failures. Using that information, a storage architect can design solutions suitable for the observed likelihood of failure.

I. Sensitivity to application workload characteristics

Our final experiments examined the sensitivity of the design solver's choices to variations in the applications' bandwidth and capacity characteristics. These experiments were conducted using the environment from the simple case study in Section IV-E and the base application characteristics described in Table II. In the capacity experiments, the capacities of all applications' datasets were scaled by a constant factor. In the

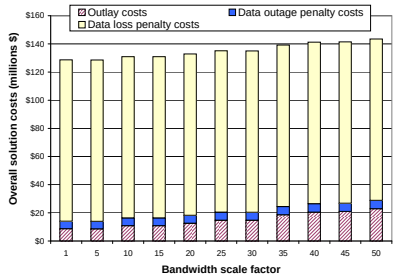

Fig. 10. Design solver's solution sensitivity to application bandwidth requirements without resource constraints

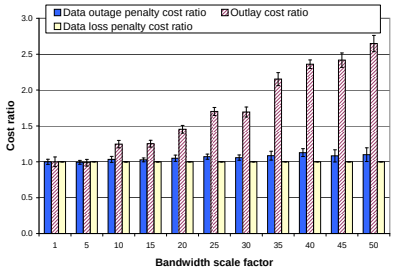

Fig. 12. Cost ratio with respect to solution cost for base bandwidth requirements (scale factor = 1) without resource constraints

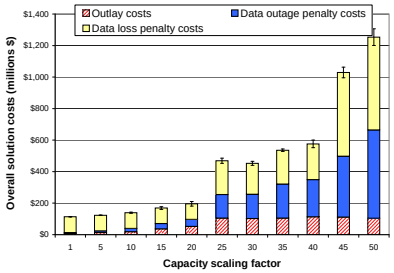

Fig. 14. Design solver's solution sensitivity to application capacity requirements with resource constraints

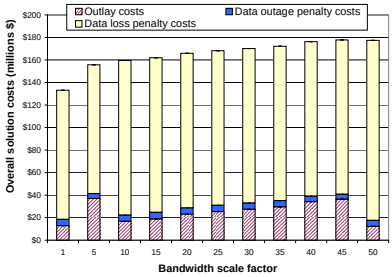

Fig. 11. Design solver's sensitivity to application bandwidth requirements with resource constraints

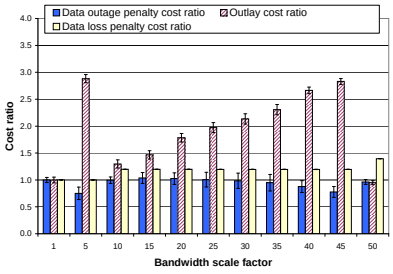

Fig. 13. Cost ratio with respect to solution cost for base bandwidth requirements (scale factor = 1) with resource constraints

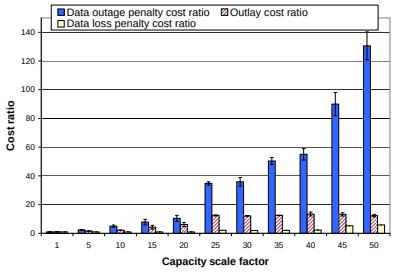

Fig. 15. Cost ratio with respect to solution cost for base capacity requirements (scale factor = 1) with resource constraints

bandwidth experiments, all of the bandwidth parameters in Table II were scaled by a constant factor. We run the design solver thirty times for each scale factor, and present the results as an average plus a 95% confidence interval.

The results from the first set of experiments, shown in Figures 10 and 12, show the behavior of the solution as application bandwidth requirements are scaled, with no constraints on the resources. As expected, we see that outlay costs increase as the peak bandwidth requirements of all applications are scaled. Data outage penalty costs increase slightly with increasing bandwidth requirements, because less network bandwidth is available. A separate set of experiments (not shown) in which capacity requirements were scaled, resulted in similar trends.

In the next set of experiments, we constrained the maximum amount of resources that could be deployed at each site. In the bandwidth scaling experiments, we constrained the amount of available network bandwidth to 64 links, totaling 1280MB/sec. In the capacity scaling experiments, we restricted the number of tape libraries to twelve; when fully populated with 24 tape drives, a tape library is capable of backing up 200 TB of data in two hours.

Figures 14 and 15 explore the sensitivity of solution cost to increasing application dataset capacity. As expected, the overall solution cost increases with increasing capacity requirements, driven by increases in data outage and data loss penalty costs. For small increases in capacity, it is more cost-effective to use the same (or similar) resources and suffer a slight increase in the outage duration (due to the need to reconstruct additional data). For slightly larger capacity increases, it is more cost-effective to add resources than to incur additional data outage penalties. We observe this behavior in Figure 14 as a piecewise linear overall cost function, with regions of scale factors 1 to 10, 15 to 20, and 25 to 50. As seen in Figure 15, outlay costs increase as a step function, while the data outage penalty costs increase for the scale factors at a single level of the step function. As resource limits are approached for the larger-capacity scale factors (e.g., 25 to 50), it may no longer be possible to maintain the same data protection choices as for lower-capacity scale factors. For example, backup windows may need to be increased from two hours to four hours, due to tape library bandwidth limitations. Those decisions result in increased recent data loss and associated penalty costs for the highest-capacity scale factors. We note that the increase in data outage and data loss penalty costs isn't a smooth linear function, because different qualitative data protection techniques (e.g., remote mirroring with failover vs. backup) are chosen for the different workloads.

Figures 11 and 13 show the sensitivity of overall solution cost to increasing application bandwidth requirements, under resource constraints. We observe three regions of behavior: scale factors 1 to 5, scale factors 10 to 45, and scale factor 50. In each region, the design solver increases network bandwidth (thus increasing outlay costs) to accommodate the increasing application bandwidth requirements, until a network bandwidth resource limit is reached. Between regions, the solver must change its data protection technique choices to ease network bandwidth requirements, in the face of increasing application bandwidth requirements. In particular, the solver must shift some of the applications from synchronous mirroring, which requires peak application update bandwidth, to asynchronous mirroring, which requires only average application update bandwidth, thus reducing network demand. That shift in storage design results in a slight increase in data loss penalty, because an asynchronous mirror is slightly out-of-date relative to a synchronous mirror. We also note that as the outlay costs increase, the outage penalties decrease (from scale factor 1 to 5 and again from scale factor 10 to 45). With the shift from synchronous mirroring to asynchronous

mirroring as the bandwidth scale factor increases, the recovery techniques have more available bandwidth to recover the failed application data, which reduces the recovery time.

Although the exact costs and piecewise linear regions in the graphs are dependent on the business requirements, workload parameters, and device parameters used in the experiments, we believe that the described trends generalize to a broader range of environments. A more comprehensive sensitivity analysis is an area for future work.

V. Related Work

Administration and deployment of storage systems are often complex. Challenges include planning the infrastructure, laying out data on the storage systems such that application performance goals are met, and planning efficient data and application failure recovery strategies such that the penalties due to a failure are minimized. Storage architects often use ad-hoc approaches, which might not provide the best solutions. Recent work addresses some of those problems by showing how to automate the design of storage systems to meet various performance goals at the lowest cost [12], [13], [14]. Minerva automates the storage design and configuration problem by decomposing it into two subproblems, which are solved separately: storage array configuration and data layout based on application workload characteristics [12]. The disk array designer handles the two issues simultaneously, using a generalized best-fit bin-packing heuristic with randomization and backtracking [14]. RAID-level selection [13] applies heuristics to choose a RAID array configuration, RAID levels, and data layout to minimize the cost while assuring that the performance requirements are met. [1], [2], [15] consider questions in the broader area of dependable storage system evaluation and design, including online and off-line data protection techniques. Keeton et al. explore methods for dependable storage design in the context of a single application and a single dependability technique [1]; the current paper considers multiple applications and combinations of techniques, which present a much more complex problem. In the area of modeling dependable storage system behavior, Keeton and Merchant presented a framework for evaluating the recovery time and recent data loss for a single application protected by a combination of techniques [2]; more recent work by their group examines how to schedule recovery operations for multiple workloads [15]. [2] considers only the dependability evaluation of an existing storage system, but does not consider how to design the system in the first place, which is the topic of this paper.

Simulation optimization is one approach by which a system designer can determine the best input parameter values from all the possible values without explicitly enumerating all possibilities. In the worst case, when the number of input parameters becomes very large, the cost to simulate all experiments becomes computationally prohibitive. The goal of simulation optimization is to minimize the computational resources that are spent while maximizing the information that is obtained through simulation. Although simulation optimization is a well-researched topic, with several surveys available on the progress of research over the past fifty years [16], [17], the characteristics of the storage design problem make it difficult to apply simulation optimization. For example,

- The objective function (total cost) for the storage design problem is non-differentiable; it cannot be expressed as an analytical function of the inputs and the constraints to the model.
- The decision variables that are being optimized can be either quantitative or qualitative. Very little literature exists to optimize systems with qualitative decision variables [16].
- The model being optimized is fairly complicated. It must be evaluated using an analytical model (data loss time) and a computer simulation model (recovery time).

The optimization strategy and evaluation algorithms must be tailored for the class of problems addressed in this paper.

Direct search methods are some of the best-known techniques for unconstrained optimization [17], [18]; they specifically address optimization problems in which the derivatives of the objective functions are not available or are not reliable. They do not make any assumptions about the underlying parameter space, but rather optimize depending upon the value of the objective function [18]. In the realm of discrete input parameters, pure combinatorial optimization problems are concerned with the efficient allocation of limited resources to meet the desired objectives [19]. Techniques for solving such problems, including linear and integer programming, expect prior knowledge of the bounds on the available resources to optimize. The number of different alternatives in the discretized space of available resources makes it computationally expensive to compute all possible combinations of allocations to solve the storage design problem using combinatorial optimization techniques. In addition, the storage design problem for data protection requires the optimization of both continuous and discrete parameters, making it significantly harder to simply use combinatorial optimization or direct search methods.

While it is interesting to determine an exact solution for a design problem, in practice it is often unnecessary, since the computer models and specifications are simplifications of reality. Often, designers are interested in good solutions that are better than those generated by very simplistic methods (system architects' past design experience). For the above reasons, researchers have developed meta-heuristics to efficiently and effectively explore the problem to obtain near-optimal solution. Some well-known meta-heuristics are general hill-climbing [20], simulated annealing [21], ant colony optimization [22], tabu search [23], and genetic algorithms [24]. Hill-climbing is a greedy approach that attempts to maximize (or minimize) the goal function by exploring the nodes neighboring the current solution. Simulated annealing (SA) is analogous to the physical process of annealing in metallurgy. At each step, the SA algorithm replaces the current solution with a neighboring solution, chosen with a probability that depends on the difference between function values and the global parameter T (for temperature), which is gradually reduced during the SA process. The solution efficiency and

effectiveness depend heavily on the parameter T. Tabu search (TS) is a generalized approach that is similar to simulated annealing, in that it keeps track of multiple generated solutions to determine the next possible potential solution.

Meta-heuristics searches are efficient in scenarios where the underlying structure of the parameter space is known [25]. Genetic algorithms (GAs) have emerged as the most popular approach to solving problems with a mix of qualitative and quantitative decision variables [16]. Dicke et al. have applied GAs to determine efficient data placement in a storage area network based on workload characteristics [10]. However, without sufficient information about the underlying problem structure or sensitivity of the decision variables, such general meta-heuristics often underperform, as shown by our comparison to the generic GA. As demonstrated by our experimental results, our technique performs better by exploring a much larger space at each local region, which is does by quickly learning the sensitivity of the decision variables.

VI. 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 in 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. That 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 [2] 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 architects today, and with a randomized search heuristic and a genetic algorithm meta-heuristic. For the examples we consider, our automated design framework consistently generates solutions that are better than the solutions provided by the competing approaches. In the case of the human heuristic, our design tool's solutions decrease overall costs by a factor of 2.

We also study the sensitivity of the design tool's solution quality to several parameters, including algorithm execution time, failure likelihood, and application bandwidth and capacity requirements. We find that the design tool's solutions converge quickly (in minutes) to the best solution. We observe that solution quality is consistent across a relatively wide range of failure likelihoods. To address increasing application bandwidth and capacity requirements, the design solver uses a combination of strategies, including employment of additional resources and shifting to less aggressive data protection choices when resource limits are met.

Our automated approach to designing storage systems to meet dependability goals provides dramatic improvements over today's manual adhoc approaches. These improvements translate into savings of millions of dollars in the expected cost of deploying and recovering the resulting storage systems.

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant Number CNS-0406351 and a generous gift from HP Labs. We would like to thank Jenny Applequist for her editorial comments.

References

[1] K. Keeton, C. Santos, D. Beyer, J. Chase, and J. Wilkes, "Designing for disasters," in *Proc. 3rd USENIX Conf. File and Storage Technologies (FAST)*, Mar. 2004, pp. 59–72.

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

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

[4] M. Ji, A. Veitch, and J. Wilkes, "Seneca: Remote mirroring done write," in *Proc. USENIX Technical Conf. (USENIX'03)*, 2003, pp. 253–268.

[5] R. R. Schulman, *Disaster Recovery Issues and Solutions*, Hitachi Data Systems white paper, Sept. 2004. [Online]. Available: http://www.hds.com/pdf/wp_117_02_disaster_recovery.pdf

[6] 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)*, Apr. 2002, pp. 259–270.

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

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

[9] W. D. Zhu, J. Cerruti, A. A. Genta, H. Koenig, H. Schiavi, and T. Talone, *IBM Content Manager Backup/Recovery and High Availability: Strategies, Options and Procedures*, IBM Redbook, Mar. 2004.

[10] E. Dicke, A. Byde, P. J. Layzell, and D. Cliff, "Using a genetic algorithm to design and improve storage area network architectures." in *Proc. Genetic and Evolutionary Computation (GECCO)*, 2004, pp. 1066–1077.

[11] Eagle Rock Alliance Ltd., "Online survey results: 2001 cost of downtime," Aug. 2001. [Online]. Available: www.contingencyplanningresearch.com/2001%20Survey.pdf

[12] G. A. Alvarez, "Minerva: An automated resource provisioning tool for large-scale storage systems," *ACM Transactions on Computer Systems*, vol. 19, no. 4, pp. 483–518, Nov. 2001.

[13] E. Anderson, R. Swaminathan, A. Veitch, G. A. Alvarez, and J. Wilkes, "Selecting RAID Levels for Disk Arrays," in *FAST '02: Proceedings of the 1st USENIX Conference on File and Storage Technologies*, 2002, pp. 189–201.

[14] E. Anderson, S. Spence, R. Swaminathan, M. Kallahalla, and Q. Wang, "Quickly finding near-optimal storage designs," *ACM Transactions on Computer Systems*, vol. 23, no. 4, pp. 337–374, 2005.

[15] K. Keeton, D. Beyer, E. Brau, A. Merchant, C. Santos, and A. Zhang, "On the road to recovery: restoring data after disasters," in *Proc. European Systems Conf. (EuroSys)*, April 2006, pp. 235–248.

[16] F. Azadivar, "Simulation optimization methodologies," in *WSC '99: Proceedings of the 31st conference on Winter Simulation*, 1999, pp. 93–100.

[17] Y. Carson and A. Maria, "Simulation Optimization: Methods And Applications," in *WSC '97: Proceedings of the 29th conference on Winter Simulation*, 1997, pp. 118–126.

[18] 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*, vol. 45, no. 3, july 2003, pp. 385–482.

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

[20] A. W. Johnson and S. H. Jacobson, "A class of convergent generalized hill climbing algorithms," *Appl. Math. Comput.*, vol. 125, no. 2-3, pp. 359–373, 2002.
[21] S. Nahar, S. Sahni, and E. Shragowitz, "Simulated annealing and combinatorial optimization," in *DAC '86: Proceedings of the 23rd ACM/IEEE Conference on Design Automation*, 1986, pp. 293–299.
[22] M. Dorigo and T. Stützle, *Ant Colony Optimization*. MIT Press, 2004.
[23] F. Glover and M. Lagun, *Tabu search*. Kluwer Academic Publishers, 1997.
[24] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Professional, January 1989.
[25] C. H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*. Dover Publications, 1998.

Shravan Gaonkar received a BE degree in Computer Engineering from the National Institute of Technology, Surathkal, Karnataka, India and an MS degree in Computer Science from the University of Illinois, Urbana-Champaign, Illinois, USA in 2000 and 2003, respectively, and is currently working toward a PhD degree at the University of Illinois. He has developed several algorithms to evaluate systems during their design phase and is currently working on new analysis tools and techniques for the evaluation of systems represented with stochastic models using simulation. His research interests include discrete simulation, storage and file systems, parameter estimation and evaluation of stochastic models. Mr. Gaonkar is also one of the developers of the Möbius tool, described at www.mobius.uiuc.edu. He can be contacted at gaonkar@ieee.org.

Kimberly Keeton received the BS degree in computer engineering and engineering and public policy from Carnegie Mellon and the MS and PhD in Computer Science from the University of California at Berkeley. She is a Senior Research Scientist at Hewlett-Packard Laboratories. Her research focuses on simplifying the design and management of enterprise information systems. She leads the data dependability research program, which determines how to automatically design data storage systems to meet customers' dependability (e.g., reliability, availability and performance) goals. She has served on numerous program committees, including ACM SIGMETRICS, USENIX Symposium on Operating Systems Design and Implementation, IEEE/IFIP International Conference on Dependable Systems and Networks, and USENIX Conference on File and Storage Technologies. She is a member of IEEE, ACM and USENIX.

Arif Merchant received the B. Tech degree in computer science from the Indian Institute of Technology at Bombay in 1984 and the Ph.D. degree in computer science from Stanford University in 1991. He is a Principal Research Scientist at Hewlett-Packard Laboratories. Prior to joining Hewlett-Packard, he worked at the IBM T. J. Watson Research Center and the NEC Computer and Communications Research Laboratories. His research interests include the modeling, design, and management of computer storage systems. He has served on several program committees for the ACM SIGMETRICS and Performance conferences and was a co-chair of the program committee for SIGMETRICS/Performance 2004. He currently serves as the Secretary/Treasurer of ACM SIGMETRICS.

William H. Sanders is a Donald Biggar Willett Professor of Engineering and the Director of the Information Trust Institute at the University of Illinois at Urbana-Champaign. He is a professor in the Department of Electrical and Computer Engineering and the Coordinated Science Laboratory. He is a Fellow of the IEEE and the ACM. He is serving on the editorial board of *Performance Evaluation* and is the Area Editor for Simulation and Modeling of Computer Systems for the *ACM Transactions on Modeling and Computer Simulation*. He is a past Chair of the IEEE Technical Committee on Fault-Tolerant Computing, and past Vice-Chair of IFIP Working Group 10.4 on Dependable Computing.

Dr. Sanders's research interests include performance/dependability evaluation, dependable computing, and reliable distributed systems. He has published more than 175 technical papers in those areas. He is a co-developer of three tools for assessing the performability of systems represented as stochastic activity networks: Möbius , METASAN, and*UltraSAN*. Möbius and *UltraSAN* have been distributed widely to industry and academia; more than 500 licenses for the tools have been issued to universities, companies, and NASA for evaluating the performance, dependability, security, and performability of a variety of systems.