A Framework for Efficient Evaluation of the Fault Tolerance of Deduplicated Storage Systems

Eric William Davis Rozier, William H. Sanders
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
Urbana, Illinois, USA
{*erozier2,whs*}*@illinois.edu*

***Abstract*—In this paper we present a framework for analyzing the fault tolerance of deduplicated storage systems. We discuss methods for building models of deduplicated storage systems by analyzing empirical data on a file category basis. We provide an algorithm for generating component-based models from this information and a specification of the storage system architecture. Given the complex nature of detailed models of deduplicated storage systems, finding a solution using traditional discrete event simulation or numerical solvers can be difficult. We introduce an algorithm which allows for a more efficient solution by exploiting the underlying structure of dependencies to decompose the model of the storage system. We present a case study of our framework for a real system. We analyze a production deduplicated storage system and propose extensions which improve fault tolerance while maintaining high storage efficiency.**

***Keywords*-storage, deduplication, reliability, simulation, decomposition**

I. Introduction

Modern storage systems have become increasingly large and complex, both because of the growth of user data, and in response to recent legislation mandating the length of time data must be stored for retrieval. In 2002, over five exabytes of data were produced [1], representing an increase of 30% from 2001. By 2007 the figure had increased to 281 exabytes, 10% more than expected because of faster growth in cameras, digital TV shipments, and other media sectors [2]. In 2010, the total data produced passed the zettabyte barrier. Forecasts put the total size of stored data for 2011 at 10 times the size for 2006, roughly 1.8 zettabytes [3]. Storing this data has become increasingly problematic. In 2007, as forecast, the amount of data created exceeded available storage for the first time [2].

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

This paper presents a framework for evaluating the fault tolerance of large-scale systems which utilize deduplication. The framework supports the analysis of existing deduplication storage profiles in order to build accurate models of the relationships between deduplicated files as these relationships can differ enough, even between categories of files on a single storage system, to shift the impact of deduplication on fault tolerance from positive to negative. Additionally this framework uses knowledge of the dependence relationships caused by deduplication and RAID to dynamically decompose the system model, based on the reward variables defined over the system, to improve solution efficiency.

Our framework consists of several parts:

- A set of component-based models of the underlying storage system.
- A model of deduplication relationships in the storage system, generated stochastically using empirical data from a real storage system.
- A method for identifying dependence relationships in the resulting model, important events which temporarily remove or return dependence relationships in a model (faults, fault propagation, and fault mitigation), and a method to automatically decompose a model based on this information to improve the efficiency of model solution.

Our framework formalizes the techniques presented in [4] for a 7TB system. In this paper we apply our framework to a model of a one petabyte storage system based on the analysis of additional data from a system similar to that presented in [4] but with a different user base, and thus different characteristics. We present strategies for improving reliability by storing additional copies of deduplicated files for a subset of the system, and show that while for some categories, deduplication has a negative impact on reliability, for others the impact is positive, demonstrating previous predictions made in [4].

A. Related Work

The cost of deduplication in terms of performance [5], [6] is well understood. Reliability studies have been much fewer in number. Since traditional deduplication keeps only

a single instance of redundant data, deduplication has the potential to magnify the negative impact of losing a data chunk [7], [8] However, due to the smaller number of disks required to store deduplicated data, deduplication also has the potential to improve reliability as well. Administrators and system architects have found understanding the data reliability of their system under deduplication to be important but extremely difficult [9].

Quantitative modeling of reliability in a deduplication system is challenging, even without taking into account the petabyte scale of storage systems. First, there are different types of faults in a storage system, including whole disk failures [10], latent sector errors (LSEs) [11], [12], and undetected disk errors [13], [14], [15]. Second, these faults can propagate due to the sharing of data chunks or chaining of files in a deduplication system. In order to correctly understand the impacts of these faults and their consequences on the reliability of our storage system, we need to accurately model both storage system faults and faults due to data deduplication. Third, it is important to note that many of the faults we wish to consider are rare compared to other events in the system, such as disk scrubbing, disk rebuilds, and I/O. Calculating the impact from rare events in a system can be computationally expensive, motivating us to find efficient ways of measuring their effect on the reliability metrics of interest.

The complexity of this problem arises from two different causes. The first is the state-space explosion problem which can make numerical solution difficult. A second issue comes from the stiffness that results from rare events. For numerical solution stiffness introduces numerical instability, making solution impractical. When simulating, stiffness increases the number of events we must process, causing a resulting increase in simulation complexity.

B. Organization

This paper is organized as follows: Section II introduces the modeling formalism used for our framework; Section III discusses the models we use for disks, reliability groups, and data deduplication relationships; Section V discusses data dependence relationships and introduces the notion of a model dependence graph (MDG); Section VI discusses methods for automatically identifying key events in an MDG; Section VII uses the methods described in Sections V and VI to decompose a model, while preserving reward variable relationships; Section VIII discusses solution methods for the decomposed model, and proves the preservation of reward metrics by these solution methods; and finally Section IX presents the application of these methods to our system of interest and discusses the results, and Section X concludes the paper.

II. Background

We present our method in the context of a generic model specification language based on the notation presented in [16]. This is intended as an alternative to presenting our results in a specific formalism, both to simplify the discussion of our techniques and to generalize our methods.

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

- *S is a finite set of state variables* $\{s_1, s_2, \ldots, s_n\}$ *that take values in* $\mathbb{N}$.
- *E is a finite set of events,* $\{e_1, e_2, \ldots, e_m\}$ *that may occur in the model.*
- $\Phi : E \times \mathbb{N}_1 \times \mathbb{N}_2 \times \ldots \times \mathbb{N}_n \to \{0, 1\}$ *is the event-enabling function specification.*
- $\Lambda : E \times \mathbb{N}_1 \times \mathbb{N}_2 \times \ldots \times \mathbb{N}_n \to (0, \infty)$ *is the transition rate function specification.*
- $\Delta : E \times \mathbb{N}_1 \times \mathbb{N}_2 \times \ldots \times \mathbb{N}_n \to \mathbb{N}_1 \times \mathbb{N}_2 \times \ldots \times \mathbb{N}_n$ *is the state variable transition function specification.*

We represent this formalism visually using circles for state variables, boxes for events, and arcs to represent the dependence of the functions Φ, Λ, and Δ on state variables and events.

In addition to specifying a model of system, one must specify the performability, availability, or dependability measures for a model. These measures are specified in terms of reward variables [17]. Reward variables are specified as a reward structure [18] and a variable type.

Definition 2. *Given model* $M = (S, E, \Phi, \Lambda, \Delta)$*, we define two reward structures: rate rewards and impulse rewards.*

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

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

Definition 3. *Let* $\Theta_M = \{\theta_0, \theta_1, \ldots\}$ *be a set of reward variables, each with reward structure* $\mathcal{R}$ *or* $\mathcal{I}$ *associated with a model* M.

The type of a reward variable determines how the reward structure is evaluated, and can be defined over an interval of time, an instant of time, or in steady state, as shown in [19], [17].

A. Instant-of-Time Variables

We refer to variables which are used to measure the behavior of a model at a particular time t as instant-of-time variables [20], [17]. Such a variable, $\theta(t)$ is defined as:

$$\theta_t = \sum_{\nu \in \mathcal{P}(S,\mathbb{N})} \mathcal{R}(\nu) \cdot I_t^{\nu} + \sum_{e \in E} \mathcal{I}(e) \cdot I_t^{e} \quad (1)$$

where

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

B. Interval-of-Time Variables

In order to calculate metrics which accumulate over some fixed interval of time, we use interval-of-time variables. Such variables accumulate reward during some interval of time, and take on the value of the total reward for the defined period [20], [17]. Given such a variable, $\theta_{[t,t+l]}$, we define it as:

$$\theta_{[t,t+l]} = \sum_{\nu \in \mathcal{P}(S,\mathbb{N})} \mathcal{R}(\nu) J_{[t,t+l]}^{\nu} + \sum_{e \in E} \mathcal{I}(e) N_{[t,t+l]}^{e} \quad (2)$$

where

- $J_{[t,t+l]}^{\nu}$ is a random variable which represents the total time the model spent in a marking such that for each $(s, n) \in \nu$, the state variable s has a value of n during the period $[t, t + l]$.
- $I_{t \to \infty}^{e}$ is an random variable which represents the number of times an event e has during the period $[t, t + l]$.

C. Time-Averaged Interval-of-Time Variables

The final type of reward variables we will consider are time-averaged interval-of-time variables. These variables quantify accumulated reward averaged over some interval of time [20], [17]. Given such a variable, $\theta'_{[t,t+l]}$ we define it as:

$$\theta'_{[t,t+l]} = \frac{\theta_{[t,t+l]}}{l} \quad (3)$$

III. System Models

A. Empirical Analysis of Deduplicated Storage System

In order to build a model of our deduplicated storage system, we examined deduplicated data stored in an enterprise backup/archive storage system that utilizes variable-chunk hashing [21], [22]. We generated a model of the deduplication reference and stored instance relationships in the manner described by [4]. The data consisted of roughly 200,000,000 deduplicated files, which were sorted into twelve categories using the algorithms presented in [4].

B. Fault Models

When modeling faults, we utilize models of traditional disk failures, LSEs and undetected disk errors (UDEs). Traditional disk failures are assumed to be non-transient and unrepairable without drive replacement. LSEs can be either transient or permanent [11]. It is important to note that even in the case of a transient LSE, a previous study of LSEs has indicated that data stored in the sector is irrevocably lost, even when the sector can later be read or written to properly [11]. In our system model, we consider LSEs to be correctable either when the disk is subsequently rebuilt due to a traditional disk failure, or upon performance of a scrub of the appropriate disk. UDEs represent silent data corruption on the disk, which is undetectable by normal means [23], [13], [14]. UDEs which manifest during writes are persistent errors that are only detectable during a read operation subsequent to the faulty write. We consider UDEs to be correctable when the disk is rebuilt because of a traditional disk failure, upon performance of a scrub of the appropriate disk, or when the error is overwritten before being read, although this type of mitigation produces parity pollution [14]. We draw our models for total disk failure, LSEs, and UDEs from those presented in [4].

In general, for a fault to manifest as a data loss error, we must experience a series of faults within a single RAID unit. How these faults manifest as errors depends on the ordering of faults and repair actions in a time line of system events. UDEs cause a different kind of error, which is largely orthogonal to RAID, by silently corrupting data which can then be served to the user.

C. Disk Model

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

D. Reliability Group Model

In order to characterize the interactions of faults in our model, we maintain a state-based model of portions of the physical disk. Given a set of disks that are grouped into an interdependent array (such as the set of disks in a RAID5 configuration, or a pair of disks that are mirrored), each stripe in the array maintains its state using a state

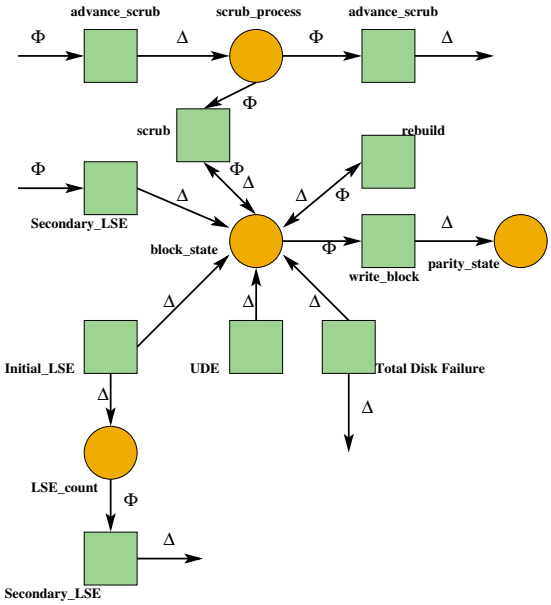


Figure 1: Block model diagram

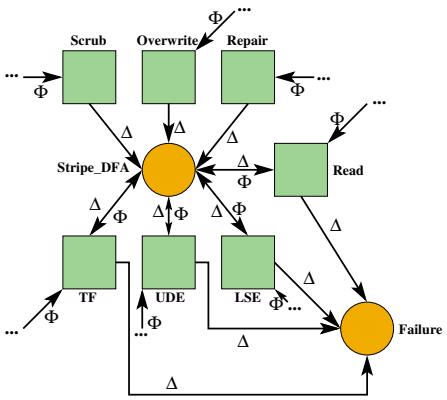
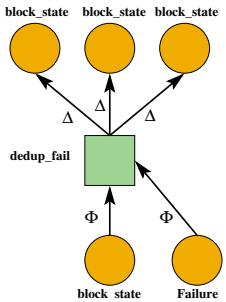


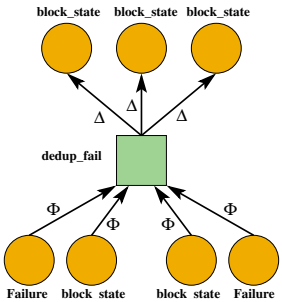
Figure 2: Stripe model diagram

machine appropriate to the number of tolerated faults the configuration can sustain without data loss.

In order to characterize the interactions of faults in our model, we maintain a state-based model of portions of the physical disk. Each stripe-based state machine is implemented by storing the stripe state in a state variable called `Stripe_DFA` (as shown in Figure 2), which takes on values $\{0, 1, 2, 3, 4, 5, 6\}$ to represent the states for our DFA. Events in this stripe model represent state transition events and have enabling conditions based on the faults present in blocks in the given stripe on the system. Another state variable, `Failure`, is set to 1 when `Stripe_DFA` has a value in $\{5, 6\}$. The DFAs maintained by stripes within our modeled system are generated automatically using knowledge of potential fault interactions and parameters that define the size of the disk array s_{array} and the number of disk faults



(a) Example model representing deduplication relationships.



(b) Example model representing deduplication relationships with 2-copy deduplication.

Figure 3: Representing deduplication in our modeling formalism

$n_{\text{tolerated faults}}$ that the array can tolerate without data loss, as defined by the array's RAID level [24].

1) Deduplication Model: We use empirical data to generate an estimate of the probability density function (pdf) for a random variable representing the number of references for a chunk in the given category c. Using this pdf, $f_c(x)$ we can generate realizations of the random variable described by $f_c(x)$, allowing us to synthetically create a deduplication system with the same statistical properties as our example system.

We encode those relationships when generating the model as dependence relationships and correlated failures that occur when an underlying block has failed, as shown in the example in Figure 3a. In the figure, when the `block_state` state variable shown at the bottom of the diagram has failed, and the `Failure` state variable indicates the failure of the stripe. An additional failure of deduplicated references occurs because of the loss of a required instance. Multi-copy deduplication modifies the underlying model as shown in Figure 3b.

IV. Dependence

In order to improve the efficiency of our solution, we attempt to exploit dependence relationships present in our model. Our hypothesis is that failures create important dependence relationships within the model, causing us to evaluate otherwise independent submodels as a larger model. We hypothesize that repair actions break these dependencies, until the next failure occurs.

A. RAID-Induced Dependence

Under normal operating conditions, the components of a storage system can be considered largely independent. Given the assumption of uniform file placement on the system (as is typical for large-scale general purpose machines), files are read and written to drives in an independent fashion. In such cases the system might be modeled more tractably as a set of independent systems, with each system solved individually. Once a failure has occurred, the entire RAID group must be

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

B. Deduplication Induced Dependence

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

C. Important Events

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

V. Understanding Dependence Relationships

We concern ourselves with four types of dependence relationships:

- *Rate dependence*: The two submodels can be said to have *rate dependence* if the transition rate function Λ of an event in one submodel is defined in terms of the state variables of the other submodels.
- *External dependence*: When an event in one submodel has an event-enabling function, Φ, defined in terms of the state or state variables of another submodel, we say the submodels feature *external dependence*.
- Δ-*dependence*: When the firing of an event changes the values of state variables in two or more otherwise independent submodels, we say that they feature Δ-*dependence*.
- *Reward dependence*: When a reward variable $\theta_i \in \Theta_M$ exists such that its reward structure is defined in terms of the state variables of two submodels, or in terms of the events of two submodels, we say they feature *reward dependence*.

Dependencies between submodels result from *direct dependencies* between events and states, or from *indirect dependencies* resulting from a sequence of direct dependencies.

A. Model Dependency Graph

We define a way to codify these relationships by constructing a *model dependency graph* (MDG). We will use an MDG in conjunction with rare events found in the model via the methods discussed in Section VI as inputs to an algorithm we introduce in Section VII, to provide a proposed decomposition of M.

Definition 4. *The MDG of a model M is defined as an undirected labeled graph, $G_M = (V, A, L)$, where V is a set of vertices composed of three subsets $V = V_S \cup V_E \cup V_\Theta$, A is a set of arcs connecting two vertices such that one vertex is always an element of the subset V_S and one vertex is always an element of the subset V_E, or one vertex is of the subset V_Θ while the other is of the set $\{V_E \cup V_S\}$, and L is a set of labels applied to elements of A from the set $\{\Phi, \Lambda, \Delta, R\}$. Let V_S denote the subset of vertices representing the state variables $S \in M$; V_E denote the subset of vertices representing the events $E \in M$, and V_Θ denote the subset of vertices representing reward variables from Θ_M.*

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

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

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

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

We represent an MDG graphically as shown in Figure 4. State variables are represented by circles, events by squares, and reward variables by diamonds. Arcs in an MDG represent dependencies. As shown in Figure 4 rate dependencies are labeled Λ (a); external dependencies are labeled with Φ (b); Δ-dependencies are labeled with Δ (c); and rate dependencies, whether impulse or rate rewards, are labeled with R (d,e).

VI. Failure, Recovery, and Mitigation Events

In order to find a way to decompose a storage model, we could require the user to specify the failure and repair actions in the model, as with the decomposition methods

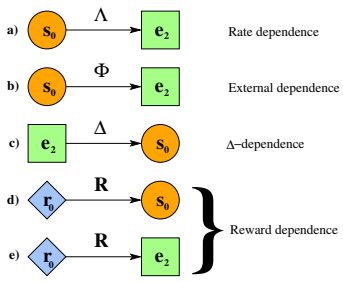


Figure 4: Two examples of near-independent submodels.

presented by [25]. Ideally, however, we wish to be able to identify these events without user input. The characteristics that set these events apart from others in the model is that they are rare.

A. Identifying Rare Events

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

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

The final, and potentially most difficult to identify, fashion in which events may be rare is when their enabling conditions defined by Φ are rare. Despite the difficulty in finding such events, they are important as they represent recovery/repair actions, among other things. It is important to identify events representing recovery, mitigation, and propagation as well as faults. The difficulty is that recovery actions have high rates compared to most failures. However, since they are not enabled unless a failure has occurred, they are dependent on a rare enabling condition. Latent sector errors also partially fall into this category. While an initial LSE is defined by a locally rare rate, studies [12] have shown that there is a period afterwords during which they become frequent. This precondition of a recent LSE creates a condition where an otherwise common event, is rare due to the state in which it is common being rare.

While calculation of all rare enabling conditions is a difficult problem, we can use domain knowledge of storage system reliability models to aid us in finding certain classes

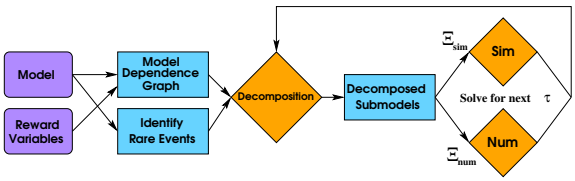


Figure 5: Overview of solution Method

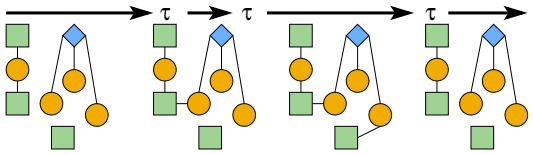


Figure 6: Model repartition after each rare event.

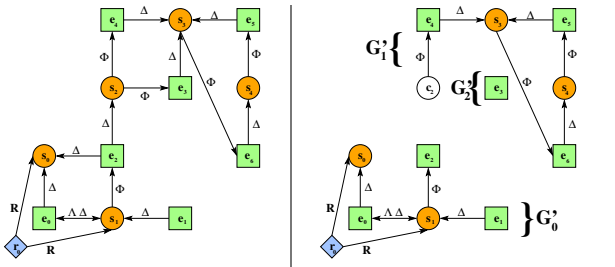
of rare enabling conditions. As we noted before, recovery actions are by necessity paired with a previous fault in the model. Without a fault, the state variables in the model can not have values such that the recovery action can fire, and so recovery actions directly depend on rare events. By analyzing the dependence relationships in our model we can identify these rare events given the assumption that our model begins with no initial faults.

B. Partitioning

We identify a certain subset $E_R \subset E$ as rare events, given some partitioning scheme, as first discussed in Section VI-A. For the models we have studied, it has been appropriate to assume a static partitioning parameter $\mu_{\max}$. We select a value for $\mu_{\max}$ based on the fact that fault events are many orders of magnitude rarer than non-fault events. Two clusters of rates were easily identified using k-means clustering with two clusters. Given a value for $\mu_{\max}$, we find those events $e_i \in E$ for which $\Lambda(e_i, q) < \mu_{max} \forall q$ and define the set of these events as E_R. This partitioning identifies those rare events that are rare due to a locally rare rate, or competition.

VII. Decomposition

In this section we present an algorithm for decomposing M, using the MDG generated from M, G_M. M will be decomposed into a set of n submodels $\Xi = \{\xi_0, \xi_1, \ldots, \xi_n\}$ that can be considered independent in the absence of the firing of a rare event. We also discuss how to repartition M using G_M after a rare event has fired, producing a new set of independent submodels, as illustrated in Figure 5. When visualizing our simulation as a time-line, as shown in Figure 6, we represent the firing of a rare event with the symbol τ. Each time a rare event fires, it results in a reevaluation of our decomposition of M, represented in Figure 5 as τ. Given a model and a set of reward variables, we generate an MDG and a set of identified rare events, E_R. Using the MDG, G_M, the set of rare events, E_R, and the current values of all state variables in the model M, we produce a decomposition of



(a) Composed Model (b) Decomposed Model

Figure 7: Example decomposition of a model dependency graph G_M to G'_M.

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

$$\Xi_{\text{Sim}} = \Xi_R \cup (\Xi_{!R} \cap \Xi_{E_R}) \quad (4)$$

$$\Xi_{\text{Num}} = \Xi_{!R} \setminus (\Xi_{!R} \cap \Xi_{ER}). \quad (5)$$

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

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

The graph G'_M that results from the application of Algorithm 1 to G_M and E_R is then used to determine if a valid partition of the model M exists for our technique. If G'_M defines multiple unconnected sub-graphs, $G'_M = \{g'_0 \cup g'_1 \cup \ldots\}$, a valid partition exists. If it does not, our technique is not applicable. The sub-graphs of G'_M correspond to the submodels in our partition Ξ. For a given sub-graph, $g'_i = (V'_i, A'_i)$, for each $v'_j \in V'_i$ such that $v'_j \in V_S$, we add the corresponding state variable to ξ_i. For each $v'_j \in V'_i$ such that $v'_j \in V_E$, we add the corresponding event to ξ_i. In addition, for each $\xi_i \in \Xi$ we restrict the definitions of $\Phi(e_j, q), \Lambda(e_j, q)$, and $\Delta(e_j, q)$ to $e_j \in \xi_i$ and $q \in \mathbb{N}_1, \mathbb{N}_2, \ldots, \mathbb{N}_{|S_{\xi_i}|}$ such that $S_{\xi_i} \in \xi_i$. An example decomposition of a small model is is shown in Figure 7.

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

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

A. Mitigation, Recovery, and Propagation Events

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

be met by the constant placeholders we use to represent the effects of a fault event's firing. Thus those events will be removed during our decomposition step.

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

B. Analyzing Reward Variable Dependencies

Reward variable dependencies prevent decomposition of otherwise independent sub-graphs by maintaining connectivity based on reward dependence and help us choose solution methods for submodels in Ξ.

Proposition 1. *In the absence of the firing of a rare event, the reward variable θ_i is independent from a submodel ξ_j if no direct dependence exists in G'_M from θ_i to a vertex in g'_j.*

Proof: If a direct dependence existed between a reward variable θ_i and a state or event in ξ_j then G'_M would have an edge connecting θ_i to a vertex in g'_j, and a path would exist. If there were an indirect dependency between θ_i and a vertex in g'_j, then a path would exist between a vertex v_k that has a direct dependency with θ_i and a vertex in g'_j. Then v_k would be a vertex in g'_j, and θ_i would have an edge connecting directly to a vertex in g'_j. ■

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

VIII. Solving the Decomposed Model

We present in this section an algorithm for hybrid simulation of decomposed models, and a discussion of complementary solution methods from the literature. Our hybrid simulation algorithm was designed to help us study the dependability characteristics of deduplicated data storage systems.

A. Hybrid Simulation of Rare-Event Decomposed Systems

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

When solving our model, we view trajectories of model execution as a time series $\tau_0 \longrightarrow \tau_1 \longrightarrow \tau_2 \longrightarrow \tau_3 \longrightarrow \ldots$ where τ_0 represents our start time, and each subsequent τ_i represents the firing of a rare event. The set Ξ_{Sim} contains all submodels that contain either a rare event or a reward dependency. The set Ξ_{Num} contains all submodels that have neither rare events nor reward dependencies. From Proposition 1, reward variable solution does not depend on Ξ_{Num}. Thus we need only solve the state occupancy probability for all submodels in Ξ_{Num} at the time of the next rare event firing. We do so by making the assumption that the submodels enter steady state between firings of rare events. This assumption seems appropriate for two reasons. The first is the high probability of a long inter-event time between rare events. The second relates to the fact that for the storage systems in which we are interested, systems tend to enter into steady state instantly in the absence of rare events. The scrub process, for example, is always in steady state; the same holds true for many recovery, propagation, or mitigation submodels. Simulation of the model M is performed using Algorithm 2, a modified version of a standard discrete event simulator.

Algorithm 2 Hybrid Simulation of M

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

The general improvement offered by this algorithm comes from the reduction of events that must be simulated in order to estimate the effect of rare events in the system. Bucklew and Radeke [26] give a general rule of thumb that in order to estimate the impact of an event with probability ρ, we must process approximately $100/\rho$ simulations. Our method seeks to reduce the number of events that must processed for each computed trajectory by eliminating those events that cannot impact Θ_M without the firing of a rare event.

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

Likewise while it is important to represent the position of the scrub process, it cannot result in mitigation of a fault until a fault is present to mitigate. By removing those events from our simulator, we improve the efficiency of our solution.

B. Required Assumptions

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

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

C. Correctness of Reward Variables

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

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

$$\theta_t = \sum_{\nu \in \mathcal{P}(S,\mathbb{N})} \mathcal{R}(\nu) \cdot I_t^{\nu} + \sum_{e \in A} \mathcal{I}(e) \cdot I_t^{e} \qquad (6)$$

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

Proposition 2. *For a given model M, a decomposed MDG G'_M, and an instant-of-time reward variable θ_t, solving for θ_t using Equation 6, and the appropriate submodel decomposition proposed by G'_M yields the same result as the original model M.*

Proof: From Proposition 1 we know that the reward variable θ_t is independent from a submodel ξ_j at time t if no direct dependence exists in G'_M from θ_t to a vertex in g'_j. Thus the solution at time t for θ_t for our decomposed submodels is the same as the solution for M. ■

Because of our method of forming an MDG, reward dependencies will exist between a variable and all state variables and events, ensuring that the submodel is decomposed in such a way that the resulting submodel contains everything necessary to evaluate θ_t.

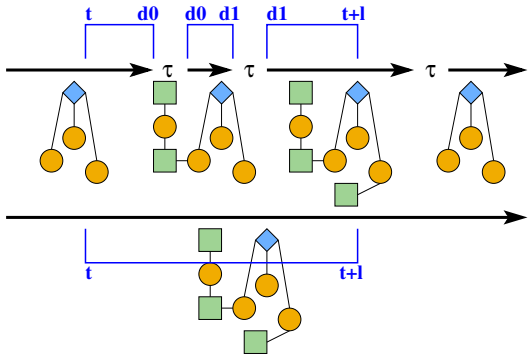


Figure 8: Comparison of instant-of-time reward variable solutions

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

$$\theta_{[t,t+l]} = \sum_{\nu \in \mathcal{P}(S,\mathbb{N})} \mathcal{R}(\nu) J^{\nu}_{[t,t+l]} + \sum_{e \in A} \mathcal{I}(e) N^{e}_{[t,t+l]} \qquad (7)$$

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

As shown in Figure 8 we have a set of n model decompositions that form intervals defined by the times $d_0, d_1, \ldots, d_{n-1}$ during the period $[t, t+l]$. For these intervals, we create $n+1$ random variables to replace $J^{\nu}_{[t,t+l]}$ and $N^{e}_{[t,t+l]}$:

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

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

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

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

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

Based on those identities, we redefine the method for calculating an interval-of-time variable for our solution method as follows:

$$Y_{[t,t+l]} = \sum_{\nu \in \mathcal{P}(S,\mathbb{N})} \sum_{d \in D} \mathcal{R}(\nu) J^{\nu}_{d} + \sum_{e \in A} \sum_{d \in D} \mathcal{I}(e) N^{e}_{d} \qquad (8)$$

The differences between that calculation and the one shown in Section II are illustrated in Figure 8. In the original model, we use a single random variable for each rate and impulse reward. Using our methods, however, we need one for each separate interval in D. The two methods are actually equivalent, however, as the sum of the new indicator variables yields the old indicator variables. The reason is that the decomposition algorithm we have presented preserves reward dependencies.

Proposition 3. *For a given model M, a set of decomposed MDGs $\hat{G'_M}$ over the interval $[t, t+l]$, and an interval-of-time reward variable $Y_{[t,t+l]}$, solving for $Y_{[t,t+l]}$ using Equation 8 and the appropriate submodel decompositions for each interval in D yields the same result as the original model M.*

Proof: Starting from the definition from Equation 8, we prove the equivalence of $Y_{[t,t+l]}$ and $\theta_{[t,t+l]}$ by construction.

$$Y_{[t,t+l]} = \sum_{\nu \in \mathcal{P}(S,\mathbb{N})} \sum_{d \in D} \mathcal{R}(\nu) J_d^\nu + \sum_{e \in A} \sum_{d \in D} \mathcal{I}(e) N_d^e \quad (9)$$

$$= \sum_{\nu \in \mathcal{P}(S,\mathbb{N})} \mathcal{R}(\nu) \sum_{d \in D} J_d^\nu + \sum_{e \in A} \mathcal{I}(e) \sum_{d \in D} N_d^e \quad (10)$$

Given Proposition 1, which states that all state variables and events required for calculating a reward variable are contained within the submodel containing the reward variable itself, we have that:

$$J^\nu_{[t,t+l]} = \sum_{d \in D} J_d^\nu \quad (11)$$

$$N^e_{[t,t+l]} = \sum_{d \in D} N_d^e \quad (12)$$

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

$$Y_{[t,t+l]} = \sum_{\nu \in \mathcal{P}(S,\mathbb{N})} \mathcal{R}(\nu) J^\nu_{[t,t+l]} + \sum_{e \in A} \mathcal{I}(e) N^e_{[t,t+l]} \quad (13)$$

$$= \theta_{[t,t+l]}. \quad (14)$$

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

■

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

$$W_{[t,t+l]} = \frac{Y_{[t,t+l]}}{l}. \quad (15)$$

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

Proposition 4. *For a given model M, a set of decomposed MDGs $\hat{G'_M}$ over the interval $[t, t+l]$, and a time-averaged interval-of-time reward variable $\theta'_{[t,t+l]}$, solving for $W_{[t,t+l]}$ using Equation 15 and the appropriate submodel decompositions for each interval in D yields the same result as the original model M.*

Proof: Given equation 14, 15 is equivalent to the definition presented in Section II. From Equation 14, we know that $Y_{[t,t+l]} = \theta_{[t,t+l]}$. Substituting $\theta_{[t,t+l]}$ for $Y_{[t,t+l]}$ in Equation 15 yields Equation 3. ■

In this section we have detailed a method to identify all dependence relationships in a model, M, using an MDG, G_M. We then detailed how to identify rare faults in the model. Using the set of identified faults, E_R, and the MDG, we showed how to enlarge the set E_R to include mitigation, repair, and propagation actions. We then showed how to use E_R with G_M to form a set of decomposed submodels, Ξ, which we then solved using Algorithm 2.

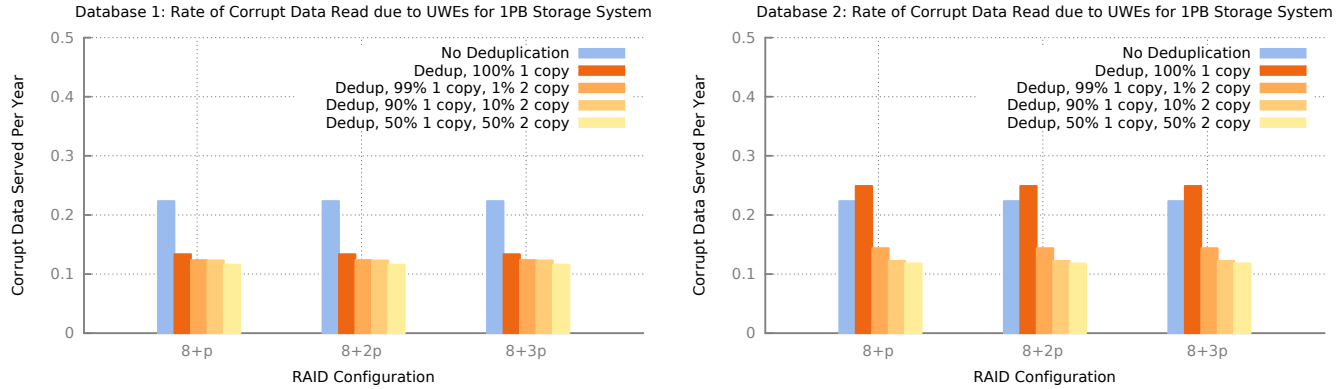
IX. Discussion

We applied our methods to a one petabyte deduplicated storage system to assess the impact of deduplication on reliability. We modeled our system under three different RAID configurations, $8+p$, $8+2p$, and $8+3p$; one, two, and three parity disks respectively. The application of our framework allowed us to automatically generate detailed models of our storage system from models of individual components in the system itself. Models of deduplication relationships were then automatically generated from empirical data. We compared the effiency of our hybrid, dependence-based, discrete event simulator to an unmodified discrete event simulator, and achieved a significant speed-up as shown in Table I.

	Speed-up
$8+p$	23.02
$8+2p$	22.53
$8+3p$	21.91

Table I: Speed-up ratio of the run-times for various RAID configurations.

In our previous work, [4], we conducted a similar study of large-scale deduplicated storage systems, but showed only a decrease in reliability due to characteristics of the data stored in the studied system. We predicted that other systems may have different results given other deduplicated different deduplication relationships. In the system studied in this paper, we confirm the predictions made in [4], once again affirming the importance of a detailed understanding of the underlying relationships in a deduplicated storage system. As shown in Figure 9, while the Archive and Database 1 file categories showed a decrease in reliability due to



(a) Corrupt data served per year due to UDEs for the Database 1 category.

(b) Corrupt data served per year due to UDEs for the Database 2 category.

Figure 10: Rates of corrupt data served for four example categories, with no deduplication, 1 copy deduplication, and 1%, 10% and 50% 2 copy deduplication.

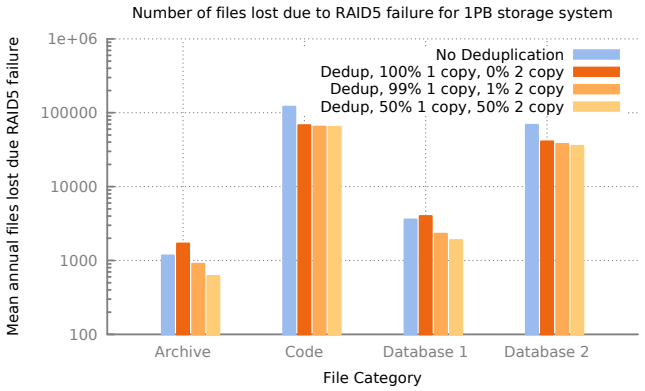


Figure 9: Annual mean files lost due to failure of $8 + p$.

deduplication, the Code and Database 2 categories showed an *increase* in reliability due to a more even distribution of references, meaning that no sets of files existed which contributed disproportionately to the impact from losses of deduplicated references. Likewise, similar results were shown for the impact of deduplication on corrupt data served due to UDEs. Figure 10 shows rates of corrupt data served per year for the categories Database 1 and Database 2. In the case of Database 1, deduplication improves reliability, where as for Database 2, reliability degrades. Results were similar for other file categories in our modeled system.

RAID implemented as $8 + 2p$ and $8 + 3p$ proved highly fault-tolerant, with few data loss events occurring, affirming that multi-copy deduplication is unnecessary for protection against additional loss due to RAID failure in one petabyte systems. Such systems, however, provide no additional protection against UDEs, however. For systems in which deduplication caused a decrease in reliability, keeping an additional copy for the most referenced 1% of files in each category was sufficient protection, though additional protection could be achieved (albeit with diminishing returns) by keeping additional copies for larger portions of a given category.

From these results, and the results we provided in [4], it is clearly important to take into account the actual deduplication characteristics of a given system when applying multi-copy deduplications strategies. With our framework it is possible to model and develop per-category multi-copy deduplication schemes to achieve the desired level of reliability, while maintaining a high level of storage efficiency. System designers need only submit to our framework the necessary component level models, RAID level, and the size of the system, along with empirical data about the footprint of the deduplicated data they wish to store. Each category in the system can then be analyzed at varying levels of multi-copy coverage. Users can enter a desired level of reliability for their system, and using our framework determine a level of multi-copy coverage which meets their goals, while obtaining more storage efficiency can be preserved than the naïve approach of assigning the same multi-copy scheme to the entire system.

X. Conclusions

This paper presents a framework for efficient solution of reliability models of large-scale storage systems utilizing deduplication. Our framework generates models from component-based templates, adds deduplication relationships derived from empirical data, and identifies dependence relationships in the generated model. These dependence relationships are used to improve the efficiency of model solution while leaving the reward measures unaffected. We demonstrate our method by solving a large-scale storage system and achieve significant speed-ups of roughly 20x when compared to unmodified discrete-event simulation. Our results show the importance of detailed models of deduplication based on file categories by showing some cat-

egories where deduplication improves reliability, and some where it decreases reliability. We show that even for a similar type of category, the impact of deduplication may not be the same.

Our results emphasize the need to generate detailed models of deduplicated systems when making design decisions. Previous work which suggested broad application of multi-copy deduplication, while effective for improving reliability, doesn't take into account the diminishing returns realized when larger percentages of the deduplicated storage system store multiple-copies of each referenced chunk. Detailed analysis has proven difficult in the past due to the complexities involved with solving large models containing rare-events, but the framework we present in this paper significantly reduces the time needed to conduct such studies.

The limit of increased reliability for systems employing these methods are the undeduplicated portions of files. Given that most storage systems are overprovisioned, or include unused hot spares to allow RAID repair, we propose that this underutilized space might be overbooked to keep additional copies of undeduplicated portions of important files, further improving reliability.

REFERENCES

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

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

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

[4] E. W. Rozier, W. H. Sanders, P. Zhou, N. Mandagere, S. M. Uttamchandani, and M. L. Yakushev, "Modeling the fault tolerance consequences of deduplication," in *Reliable Distributed Systems (SRDS), 2011 30th IEEE Symposium on*, oct. 2011, pp. 75 –84.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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