SimEDC: A Simulator for the Reliability Analysis of Erasure-Coded Data Centers

Mi Zhang, Shujie Han, and Patrick P. C. Lee

Abstract—Modern data centers employ erasure coding to protect data storage against failures. Given the hierarchical nature of data centers, characterizing the effects of erasure coding and redundancy placement on the reliability of erasure-coded data centers is critical yet unexplored. This paper presents a discrete-event simulator called SIMEDC, which enables us to conduct a comprehensive simulation analysis of reliability on erasure-coded data centers. SIMEDC reports reliability metrics of an erasure-coded data center based on the configurable inputs of the data center topology, erasure codes, redundancy placement, and failure/repair patterns of different subsystems obtained from statistical models or production traces. It can further accelerate the simulation analysis via importance sampling. Our simulation analysis based on SIMEDC shows that placing erasure-coded data in fewer racks generally improves reliability by reducing cross-rack repair traffic, even though it sacrifices rack-level fault tolerance in the face of correlated failures.

1 INTRODUCTION

Modern data centers enable large-scale storage management for cloud computing services and big data analytics. However, extensive field measurements have shown that failures, either transient or permanent, are commonplace in data centers [10], [27], [32]. To protect data storage against failures, modern data centers (e.g., [10], [18], [23]) adopt *erasure coding* to add redundancy into data storage, so that any unavailable or lost data can be recovered from other available redundant data. Erasure coding provides a storage-efficient way to construct redundancy in data storage, and provably incurs much lower storage redundancy than simple replication [37]. Its storage efficiency over replication also implies significant savings in operational costs, power, and footprints [18]. On the other hand, erasure coding has a drawback of incurring high repair penalty, as the repair of any lost erasure-coded data will trigger a transfer of much more available data than the actual amount of lost data. The amount of repair traffic can reach hundreds of terabytes per day in production data centers and overwhelm the bandwidth resources for foreground applications [27].

Thus, extensive studies in the literature focus on minimizing the repair traffic in erasure-coded storage (e.g., [8], [18], [21], [22], [28], [30], [35]). In particular, the repair problem in erasure-coded data centers poses a unique research challenge due to the *hierarchical* data center architecture, in which multiple nodes (or servers) are grouped in racks, and the cross-rack bandwidth is typically much more limited than the inner-rack bandwidth [2], [5]. This leads to two possible redundancy placement schemes. Most studies (e.g., [10], [18], [23], [28]) adopt *flat placement*, in which erasure-coded data is distributed across distinct nodes, each of which is located in a distinct rack, to maximize the tolerance against rack failures. However, the repair of any lost data in flat placement inevitably triggers cross-rack transfer of available data. On the other hand, recent studies [16], [17], [33] argue that rack failures are much rarer than node failures [7], [10], and hence advocate *hierarchical placement*, in which erasure-coded data is distributed across fewer racks, or equivalently multiple nodes per rack, to trade rack-level fault tolerance for the reduction of cross-rack repair traffic. By enabling partial repair operations within each rack, the cross-rack repair traffic can be provably minimized [16], with over 40% reduction of the minimum repair traffic achievable by the classical minimum-storage regenerating codes [8].

From the perspectives of reliability analysis, the erasure coding configuration and the redundancy placement in erasure-coded data centers pose new reliability issues: (1) How much can the reduction of cross-rack repair traffic improve reliability? (2) What is the reliability trade-off of sacrificing rack-level fault tolerance for reduced cross-rack repair traffic? (3) How does the reliability of an erasure-coded data center vary subject to more complicated failure patterns? While the literature is rich of modeling- or simulation-based reliability studies on storage systems, the reliability analysis that specifically takes into account the hierarchical nature of erasure-coded data centers remains largely unexplored.

In this paper, we present a comprehensive simulation study on the reliability of an erasure-coded data center. Our key contributions are two-fold:

- We build SIMEDC, a discrete-event simulator that characterizes the reliability of an erasure-coded data center. It is designed to be comprehensive by accounting for various factors as inputs, including the data center topology, erasure codes (e.g., the classical Reed-Solomon codes [29], and the recently proposed Local Reconstruction Codes [18] and Double Regenerating Codes [16], [17]), redundancy placement (i.e., flat or hierarchical), as well as failure/repair patterns of different subsystems derived from either statistical models or production traces. It reports different reliability metrics that capture the durability and availability of an erasure-coded data center. Furthermore, it can adopt *Importance Sampling* [12], [20], [24] to accelerate the

- *A preliminary version of this paper appeared in [38]. In this extended version, we extend* SIMEDC *with importance sampling to accelerate the simulation analysis.*
- *M. Zhang, S. Han, and P. Lee are with the Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong (E-mails: millyz0204@gmail.com, {sjhan,pclee}@cse.cuhk.edu.hk).*

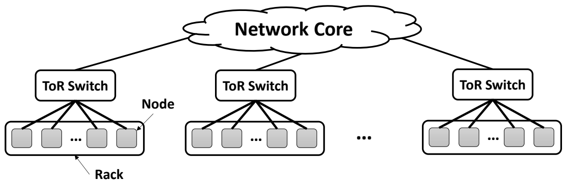


Fig. 1. Architecture of a hierarchical data center.

simulation process.

- We conduct extensive reliability analysis using SIMEDC. We find that hierarchical placement generally achieves higher reliability than flat placement due to the reduction of cross-rack repair traffic, even though its reliability degrades in the presence of correlated failures. We also observe similar behaviors based on production traces collected at Los Alamos National Laboratory [31].

The source code of our SIMEDC implementation is available at **http://adslab.cse.cuhk.edu.hk/software/simedc**.

2 Background and Problem

2.1 Data Center Architecture

We consider a hierarchical data center, as shown in Figure 1, that comprises multiple *racks*, each of which holds a number of commodity machines called *nodes* (or servers). Each node is further attached with one or multiple *disks* that provide storage space. Nodes within the same rack are interconnected by a *top-of-rack (ToR) switch*, and the racks are interconnected by a *network core* that is composed of layers of aggregation and core switches [3]. Such a hierarchical data center architecture is also considered in previous work [5], [16], [33].

We assume that the data transfer performance of a hierarchical data center is bottlenecked by the available cross-rack bandwidth. In practice, the cross-rack bandwidth is much more constrained than the inner-rack bandwidth due to oversubscription in the network core [2], [5]. Although data transfers can be slowed down by disk I/Os, in practical data centers, each node can be attached with an array of multiple commodity disks to achieve much higher sequential disk I/O throughput than the network speed [5]. Furthermore, in the context of storage reliability, the bandwidth allocated for storage-repair tasks are often throttled [18], [35], which further limits the reconstruction performance of failed data and hence degrades the overall storage reliability. Thus, how the cross-rack bandwidth affects storage reliability is one key topic of our reliability analysis.

2.2 Failure Model

Practical data centers are susceptible to failures. In our analysis, we focus on failures occurring at three levels of subsystems: racks, nodes, and disks. Failures can be as either *transient*, in which a subsystem is only temporarily unavailable without causing actual data loss (e.g., due to network disconnection, reboots, or maintenance), or *permanent*, in which a subsystem failure can lead to permanent data loss (e.g., due to disk crashes).

Failures can be further classified as *independent*, in which subsystems fail independently, or *correlated*, in which a number of subsystems fail simultaneously due to a common failure event. Correlated failures are more severe than independent failures. For example, when a ToR switch of a rack is broken, all nodes within the rack will become temporarily unavailable. One common type of failures is power outages, in which a significant fraction of nodes (up to 1%) will crash after a power-on restart and cause permanent data loss [4], [34].

Our work considers the following failure events:

- *Disk failures:* We focus on permanent disk failures, in which all data on a failed disk is lost. For simplicity, we currently do not consider latent sector errors that damage only partial data of a disk, as their severity heavily depends on the complicated data layout on the whole disk.
- *Node failures:* We consider both transient and permanent node failures. In the former, all disks attached to a failed node are only temporarily unavailable without data loss, while in the latter, we assume that the data stored on all disks is permanently lost.
- *Rack failures:* We only consider transient rack failures, in which the data of all nodes within a failed rack becomes unavailable, yet there is no data loss.
- *Correlated failures:* We treat a rack as the largest failure domain, such that a correlated failure brings down a fraction of nodes within a rack. We focus on permanent correlated failures, such that the failed nodes incur data loss (e.g., due to power outages [4], [34]).

2.3 Erasure Coding

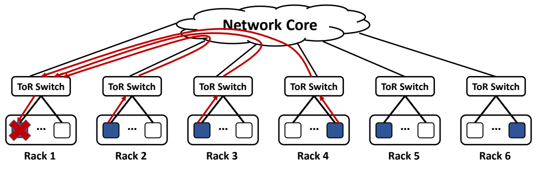
An erasure code is often constructed by two parameters n and k, where $k < n$. Suppose that a data center organizes data as fixed-size units called *chunks*. Then for every k original uncoded chunks, an erasure code encodes them into n coded chunks of the same size, such that the collection of the n coded chunks is called a *stripe*. A data center typically contains multiple stripes that are independently encoded. An erasure code is said to be *Maximum Distance Separable (MDS)* if any k out of the n coded chunks of a stripe can reconstruct the original k uncoded chunks (i.e., an MDS code can tolerate the failures of up to $n - k$ chunks), while the amount of storage redundancy is minimum (i.e., storage-optimal).

Most erasure codes deployed in practice are *systematic* codes, meaning that the original data is kept in storage after encoding. That is, for (n, k) codes, k of the n chunks of each stripe are exactly the original k uncoded chunks that can be directly accessed. In our analysis, we do not differentiate between uncoded and coded chunks, and we focus on measuring the durability and availability of all chunks stored in a data center (see Section 3). We use "chunks" to collectively refer to both uncoded and coded chunks if the context is clear.

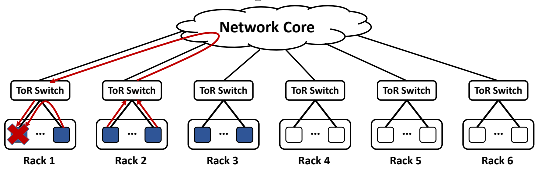
Erasure coding incurs high repair penalty as it needs to retrieve multiple chunks in order to repair a failed chunk that is unavailable or lost. We define the *repair traffic* as the amount of information retrieved for a repair operation. For example, for (n, k) MDS codes, a standard approach of repairing a failed chunk is to retrieve k available chunks of the same stripe (i.e., the repair traffic is k chunks). Since the most common failure scenario in practice [18], [27] is a *single* failure (i.e., each stripe has only one single failed chunk), many erasure codes have been proposed to improve the repair performance by reducing the repair traffic for a single-chunk repair. In this paper, we focus on three representative erasure codes that incur different amounts of repair traffic for a single-chunk repair:

- *Reed-Solomon (RS) codes:* RS codes [29] are the classical MDS codes that have been widely deployed in modern data centers [10], [35]. RS codes follow the standard repair approach of MDS codes. That is, given (n, k), the repair traffic of a single-chunk repair in RS codes is k chunks.
- *Local Reconstruction Codes (LRC):* Some erasure codes (e.g., [18], [30]) exploit locality to reduce repair traffic. In this paper, we focus on Azure's LRC [18]. It divides k uncoded chunks of a stripe into l local groups (assuming that k is divisible by l) and creates one local coded chunk for each local group, and additionally creates $n-k-l$ global coded chunks by encoding all k uncoded chunks. Given (n, k, l), the repair traffic of repairing an uncoded chunk or a local coded chunk is $\frac{k}{l}$ chunks (retrieved from the same local group), while that of repairing a global coded chunk is k chunks (retrieved from the same stripe). Note that LRC is non-MDS: even though each stripe has $n-k$ additional coded chunks, LRC cannot tolerate all possible failures of $n-k$ chunks. For example, LRC(16,12,2) cannot tolerate the failures of four chunks in the same local group, even though it has four parity chunks in total in each stripe.
- *Double Regenerating Codes (DRC):* Some studies (e.g., [16], [33]) focus on reducing the cross-rack repair traffic in an erasure-coded data center by storing multiple chunks in one rack (Section 2.4 explains the details of chunk placement). In this paper, we focus on DRC [16], which provably minimizes the cross-rack repair traffic. It distributes n chunks of a stripe across r distinct racks, where n is divisible by r, and each rack holds $\frac{n}{r}$ chunks in different nodes within the rack. In a single-chunk repair, DRC exploits a two-phase approach: it first performs partial repairs by selecting a node (called *relayer*) to encode the available chunks of the same stripe within each rack, and then re-encodes the encoded chunks from multiple relayer nodes across different racks to reconstruct the failed chunk. Like RS codes, DRC is also MDS with optimal storage redundancy. Note that DRC can be viewed as an extension to the classical *minimum-storage regenerating (MSR) codes* [8], which minimize the repair traffic for a single-chunk repair under the minimum storage redundancy. If we set $r = n$ (i.e., one chunk per rack), DRC achieves the same minimum repair traffic, given by $\frac{n-1}{n-k}$ chunks, as MSR codes. In general, given (n, k, r), the minimum cross-rack repair traffic of DRC is $\frac{r-1}{r-\lfloor kr/n \rfloor}$ chunks [16].

If a stripe contains more than one failed chunk but no more than $n-k$ failed chunks, we resort to the standard repair approach by retrieving k available chunks of the same stripe (note that the repair of LRC may fail as it is non-MDS). Specifically, for a failed chunk, if it is the only failed chunk in a stripe, the repair traffic follows the improved single-chunk repair approach of the given erasure code; otherwise, the repair traffic is k chunks. We assume that we repair one failed chunk of a stripe at a time, and we do not consider repairing multiple failed chunks simultaneously in one stripe.



(a) Flat placement



(b) Hierarchical placement

Fig. 2. Example of repairing a failed chunk under flat placement and hierarchical placement, using RS codes with $n = 6$ and $k = 3$. The nodes that hold the chunks of the same stripe are represented in dark color. In flat placement, the six chunks of a stripe reside in six racks, while in hierarchical placement, the six chunks of a stripe reside in three racks.

2.4 Chunk Placement

To tolerate node or rack failures, erasure coding places the chunks of each stripe in different nodes and racks. We consider two chunk placement schemes for each stripe of n chunks:

- *Flat placement:* The n chunks of a stripe are stored in n different nodes that reside in n distinct racks (i.e., one chunk per rack). This provides the maximum fault tolerance against both node and rack failures. The trade-off is that repairing a failed chunk must retrieve available chunks from other racks, thereby incurring a significant amount of cross-rack repair traffic. Flat placement is commonly used in production data centers [10], [18], [23], [28].
- *Hierarchical placement:* The n chunks of a stripe are stored in n different nodes that reside in $r < n$ distinct racks, each of which has n/r chunks, assuming that n is divisible by r. This reduces the cross-rack repair traffic, as repairing any failed chunk can leverage the available chunks within the same rack. The trade-off is that fewer rack failures can be tolerated than flat placement.

Note that RS codes and LRC can adopt both flat and hierarchical placements. Figure 2 shows an example of how hierarchical placement incurs less cross-rack repair traffic than flat placement, using RS codes with $n = 6$ and $k = 3$. Suppose that a node wants to reconstruct a failed chunk in its local storage. In flat placement (see Figure 2(a)), each of the six chunks of a stripe is placed in a distinct rack. Repairing the failed chunk will retrieve three chunks across racks. On the other hand, in hierarchical placement (see Figure 2(b)), we can place two chunks in a rack. Repairing the failed chunk can retrieve one chunk from the same rack and two more chunks from other racks, so the cross-rack repair traffic is reduced to two chunks. DRC specifically exploits hierarchical placement to minimize the cross-rack repair traffic.

In practice, the numbers of nodes and racks are much larger than the stripe size n. Thus, we adopt the notion of *declustered* placement [36] to place n chunks: for flat placement, we randomly select n racks from all available racks, followed by randomly selecting one node from all available

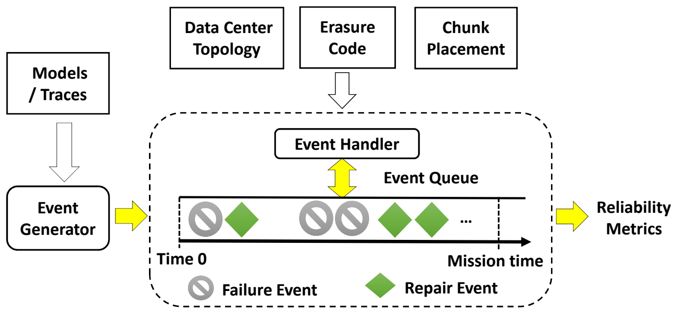


Fig. 3. SIMEDC architecture.

nodes within the same rack; for hierarchical placement, we again randomly select r racks and n/r nodes per rack. Thus, when we repair a failed node that stores the failed chunks of multiple stripes, we can retrieve available chunks from all available nodes and racks in the whole data center, thereby better harnessing parallelism to improve repair performance and hence storage reliability [36]. Based on declustered placement, our goal is to study the reliability trade-off between flat and hierarchical placements for different erasure codes.

3 SIMEDC DESIGN

We present SIMEDC, a discrete-event simulator that characterizes the reliability of an erasure-coded data center via simulation. SIMEDC builds on the High-Fidelity Reliability Simulator (HFRS) [12], which is written in Python and originally designed for the reliability simulation of a monolithic disk array. SIMEDC extends HFRS to support various erasure codes and chunk placement schemes in a hierarchical data center with the cross-rack network bandwidth constraint.

3.1 Architectural Overview

Figure 3 shows the SIMEDC architecture. At a high level, SIMEDC performs the reliability simulation over a sufficiently large number of iterations. In each iteration, it takes the data center topology, erasure code construction, and chunk placement as inputs for initialization. It records the chunk positions for a number of stripes, specified by the storage capacity that is simulated, across different nodes and racks; note that the chunk positions vary across iterations. It generates a sequence of failure and repair events, and processes them in chronological order until a failure event triggers data loss or a pre-specified system mission time (e.g., 10 years) is reached. It then outputs a set of reliability metrics for the iteration. Finally, it reports the reliability metrics averaged over all iterations.

SIMEDC allows to generate events from two sources, either statistical models for the failure and repair behaviors, or event traces that record the failure and repair events in a production data center. Both sources of events can be specified as inputs to SIMEDC before the simulation starts.

3.2 Reliability Metrics

SIMEDC measures three reliability metrics:

- **Probability of data loss (PDL):** It measures the likelihood that a data center experiences the unrecoverable loss of any chunk (i.e., the number of permanently failed chunks in an erasure-coded stripe exceeds the tolerable limit) over a mission time.
- **Normalized magnitude of data loss (NOMDL):** It is proposed by Greenan *et al.* [14] to measure the expected amount of data loss (in bytes) normalized to the storage capacity. It has several key properties that arguably improve existing reliability metrics [14].
- **Blocked ratio (BR):** It measures the fraction of time that a chunk cannot be directly accessed due to the transient or permanent failures of the subsystem that holds the chunk. Note that such an inaccessible chunk may still be recoverable from other available chunks of the same stripe in other subsystems, but it incurs extra overhead of reconstructing the chunk. Thus, the BR models the duration when a chunk cannot be directly accessed in normal mode (i.e., without failures).

In Section 3.4, we elaborate how these metrics are computed in our implementation. Note that both PDL and NOMDL are used to measure *durability*, while the BR is used to measure *availability*. A data center achieves good reliability if the values of the metrics are small.

3.3 Event Handling

Each failure or repair event in SIMEDC is represented in a tuple of three fields: (1) the timestamp when the event occurs, (2) the event type, and (3) the subsystem associated with the event. SIMEDC stores all events in an *event queue*, which is implemented as a priority queue that returns the event with the smallest timestamp for the *event handler* to process accordingly (see Figure 3). We handle permanent and transient failures separately, and consider four event types: (1) a permanent failure, (2) a transient failure, (3) a permanent failure repair, and (4) a transient failure repair.

Failure handling: Each subsystem (i.e., rack, node, or disk) is associated with one of the three states during the simulation: (1) `normal` (i.e., no failure occurs), (2) `unavailable` (i.e., a transient failure occurs), and (3) `crashed` (i.e., a permanent failure occurs). In terms of severity, `normal` is the least severe, `unavailable` is the middle, and `crashed` is the most severe. We assume that if a subsystem fails, its state will be updated only if the state becomes more severe. That is, a `normal` or `unavailable` state becomes `crashed` for a permanent failure, or a `normal` state becomes `unavailable` for a transient failure; however, a `crashed` state remains unchanged. Also, all its descendant subsystems in a hierarchical data center will inherit the same state that is more severe. That is, if a node is `crashed`, then all the disks attached to the node are also `crashed`; if a rack (resp. node) is `unavailable`, then all the nodes and disks within the rack (resp. all attached disks) are also `unavailable` if they are originally `normal`.

SIMEDC processes failure events (see Section 2.2) from the event queue. Upon receiving a permanent failure event, it checks if every chunk stored in the crashed subsystem can be repaired by a sufficient number of available chunks of the same stripe. If not, it concludes that there is data loss and returns the reliability metrics for the current iteration. If there is no data loss or a transient failure event is received, SIMEDC triggers a repair event of the same type (i.e., permanent or transient) for the failed subsystem and inserts the event into the event queue for later repair handling.

Repair handling: Before inserting a repair event into the event queue, SIMEDC computes the repair time needed to repair a permanent or transient failure. For a permanent failure, the repair time is calculated by dividing the total amount of cross-rack repair traffic for all failed chunks by the available cross-rack bandwidth. For a transient failure, the repair time is determined by either the statistical models or the event traces for the corresponding subsystem (see Section 3.1).

One subtlety is that when a permanent failure occurs, a failed chunk may not be able to be repaired immediately, since other subsystems associated with the same stripe are currently under transient failures and there are insufficient available chunks for repairing the failed chunk (although there is no data loss). Thus, if we find that the failed chunk cannot be repaired immediately due to too many transient failures in the same stripe, we add the repair time for the failed chunk by the amount of time until there are sufficient available chunks for the repair, by checking the repair times of the repair events of the related transient failures in the event queue.

To simplify repair handling, we do not consider how to optimally schedule the repairs of multiple failed chunks of a permanently failed subsystem to minimize the total repair time. In addition, if a stripe that is currently under repair has an additional failed chunk, we do not modify the repair time of any already triggered repair event. Our observation is that each stripe has at most one failed chunk in most cases throughout the mission time in our evaluation (see Section 5); in fact, field studies also confirm that single-chunk repairs dominate in practice [18], [27]. Thus, the repair time of a permanently failed chunk is mostly determined by the cross-rack repair traffic incurred for a single-chunk repair.

When a repair event is received from the event queue, SIMEDC updates the state of the associated subsystem to the normal state. In addition, if any descendant subsystem has the same failure type, we also update its state to normal (although they may fail due to different reasons). For example, if a `crashed` (resp. `unavailable`) node is repaired, any of its associated disks that is `crashed` (resp. `unavailable`) is also repaired and its state becomes `normal`. Finally, SIMEDC creates the next failure event of the same type (i.e., permanent or transient) for the subsystem and inserts the event into the event queue for later failure handling.

Algorithm 1 SIMEDC

```
1: procedure MAIN
2:     for $i = 1$ to $X$ do
3:         (PDL$_i$, NOMDL$_i$, BR$_i$) $\leftarrow$ SIMULATE
4:     end for
5:     return $\frac{1}{X}\sum_{i=1}^{X}$(PDL$_i$, NOMDL$_i$, BR$_i$)
6: end procedure

7: function SIMULATE
8:     Initialize the erasure-coded data center and set $L = 1$
9:     Create $Q_f$ and $Q_r$
10:    Generate the first failure event for each subsystem
11:    Push all failure events to $Q_f$
12:    while true do
13:        (time $t$, type $y$, subsystem $s$) $\leftarrow$ GET_NEXT_EVENT
14:        if $t > T$ then
15:            return (0, 0, BR)
16:        end if
17:        if $y$ is a permanent failure then
18:            if data loss occurs then
19:                return (L, NOMDL, BR)
20:            else
21:                $s$.state $\leftarrow$ crashed
22:                for each $s$'s descendant $s_d$ do
23:                    $s_d$.state $\leftarrow$ crashed
24:                end for
25:                $t_R \leftarrow \frac{\text{Cross-rack repair traffic}}{\text{Cross-rack bandwidth}}$
26:                $Q_r$.Push($t + t_R$, permanent failure repair, $s$)
27:            end if
28:        else if $y$ is a transient failure then
29:            if $s$.state == normal then
30:                $s$.state $\leftarrow$ unavailable
31:                for each $s$'s normal descendant $s_d$ do
32:                    $s_d$.state $\leftarrow$ unavailable
33:                end for
34:            end if
35:            $t_R \leftarrow$ Repair time of $s$ from models or traces
36:            $Q_r$.Push($t + t_R$, transient failure repair, $s$)
37:        else if $y$ is a permanent failure repair then
38:            if $s$.state == crashed then
39:                $s$.state $\leftarrow$ normal
40:                for each $s$'s crashed descendant $s_d$ do
41:                    $s_d$.state $\leftarrow$ normal
42:                end for
43:            end if
44:            $t_F \leftarrow$ time to next permanent failure
45:            $Q_f$.Push($t + t_F$, permanent failure, $s$)
46:        else if $y$ is a transient failure repair then
47:            if $s$.state == unavailable then
48:                $s$.state $\leftarrow$ normal
49:                for each $s$'s unavailable descendant $s_d$ do
50:                    $s_d$.state $\leftarrow$ normal
51:                end for
52:            end if
53:            $t_F \leftarrow$ time to next transient failure
54:            $Q_f$.Push($t + t_F$, transient failure, $s$)
55:        end if
56:    end while
57: end function
```

3.4 Putting It All Together

Algorithm details: Algorithm 1 shows the pseudo-code of the workflow of SIMEDC. The MAIN procedure (Lines 1-6) executes the reliability simulation function SIMULATE over a number of iterations X, where X is tunable (see below).

In each iteration, SIMULATE first performs initialization (Line 8), in which it initializes the data center topology, erasure coding scheme, and chunk placement. Also, it defines the likelihood ratio L, which is used in importance sampling (see Section 4). We now set $L = 1$ throughout the algorithm.

SIMULATE creates two queues Q_f and Q_r (Line 9), which store the failure events and repair events, respectively. It also generates the first failure event for each subsystem based on the specified failure distributions and adds them to Q_f (Line 11). It then extracts one event from either Q_f or Q_r through the GET_NEXT_EVENT function (see Algorithm 2). If the system is normal (i.e., no failure exists), GET_NEXT_EVENT returns the first failure event to process; otherwise, it returns either the next failure event or the next repair event, depending on which one has a smaller

Algorithm 2 Get_next_event

```
1: function GET_NEXT_EVENT
2:     if the whole system is normal then
3:         (time $t$, type $y$, subsystem $s$) ← $Q_f$.Pop
4:     else
5:         if timestamp of next failure event < timestamp of next repair event then
6:             (time $t$, type $y$, subsystem $s$) ← $Q_f$.Pop
7:         else
8:             (time $t$, type $y$, subsystem $s$) ← $Q_r$.Pop
9:         end if
10:    end if
11:    return (time $t$, type $y$, subsystem $s$)
12: end function
```

timestamp. While we can maintain a single event queue as in Figure 3 to arrange all failure and repair events in the order of their timestamps, separating the failure and repair events into two queues simplifies our later extension for importance sampling (see Section 4).

SIMULATE terminates if the event time exceeds the mission time T (Line 15); otherwise, it processes the event according to one of the four event types: permanent failure (Lines 18-27), transient failure (Lines 29-36), permanent failure repair (Lines 38-45), and transient failure repair (Lines 47-54). Each failure (resp. repair) event will trigger the next repair (resp. failure) event of the same type (i.e., permanent or transient). This ensures that each subsystem must have exactly one pending failure or repair event for both permanent and transient types.

SIMULATE returns a tuple of PDL, NOMDL, and BR in each iteration. For the PDL, it is 0 if there is no data loss (Line 15), or $L = 1$ otherwise (Line 19). For the NOMDL, it is 0 if there is no data loss (Line 15); otherwise, it is given by the total number of chunks that are unrecoverable divided by the total number of chunks stored in the data center (Line 19). For the BR, it is computed as the fraction of time that a chunk is in the `normal` state over the mission time, averaged over all chunks stored in the data center.

Note that SIMULATE introduces random factors in different places in each iteration, including: the chunk placement in a data center (Line 8) and the generation of failure and repair events according to the statistical models (Lines 10, 35, 45, and 54). Thus, the returned results of SIMULATE are different across iterations.

Configuring the number of iterations: One key question is how to configure the "right" number of iterations X in our simulation. A large X improves simulation accuracy, but incurs a significantly long simulation time. In SIMEDC, we use the relative error (RE) of the measured PDL to configure the number of iterations. Suppose that we choose the 95% confidence interval. Then the RE of the currently measured PDL (denoted by p) is given by:

$$\mathrm{RE} = \frac{1.96}{p}\sqrt{\frac{p(1-p)}{X-1}}. \tag{1}$$

Our goal is to run a sufficient number of iterations such that RE is less than 20% [12]. Initially, we set $X = 1{,}000$ and obtain p. If the RE is less than 20%, we stop the simulation, and return p as the PDL as well as both measured NOMDL and BR. Otherwise, we compute a new X from Equation (1) with RE = 20% and the current value of p. We then run more iterations until the total number of executed iterations is equal to the new X. We check the RE again and add more iterations if needed.

To control the simulation time, we set the maximum total number of iterations to be executed as 20,000, and stop the simulation anyway if the maximum number of iterations is reached. The main limitation is that for the erasure codes that are highly reliable (e.g., the codes with high redundancy or small repair traffic), the measured PDL may be too small such that the RE remains high, or we may not even observe a data loss event when the maximum total number of iterations is reached [12]. In such cases, the reliability results should not be fully trusted, although they can provide indicators that the storage system is already very reliable.

Parallelizing simulation: Our simulation is embarrassingly parallel as the iterations are independent. Thus, we further accelerate the whole simulation through parallelization. Specifically, we split the X iterations of SIMULATE in Algorithm 1 into multiple subsets, each of which is executed by a standalone process. We distribute the processes across multiple CPU cores in multiple machines. Finally, we compute the average results from all processes.

4 IMPORTANCE SAMPLING

SIMEDC thus far realizes discrete-event simulation by simulating the occurrence of each event one by one. In this section, we show how SIMEDC speeds up the simulation workflow through importance sampling. Note that importance sampling is a well-known technique for realizing accelerated simulation analysis for rare events (e.g., failures in highly reliable systems). We refer readers to [12], [24] for the detailed explanation on importance sampling. We do not claim the novelty of the technique itself, but instead our goal is to demonstrate how SIMEDC can build on importance sampling to speed up the simulation analysis specifically for an erasure-coded data center with a hierarchical topology.

Background: The idea of importance sampling is to estimate some rarely occurred properties of a probability distribution by sampling from another probability distribution that increases the occurrences of the properties. In the case of SIMEDC, the property of interest refers to the permanent failures. By increasing the occurrences of permanent failure events, we can measure the reliability of a data center in accelerated mode.

As a case study, we implement importance sampling based on *uniformization-based balanced failure biasing* [12], [20], [24]. Specifically, through uniformization, we sample permanent failure events from a homogeneous Poisson process with arrival rate β to match a non-homogeneous Poisson process with arrival rate $\lambda(t)$, such that we set $\beta \geq \lambda(t)$ for all t to "thin" the homogeneous process to provide points for the non-homogeneous process [20]. Thus, we draw a permanent failure event from the homogeneous Poisson process at time t with probability $\frac{\lambda(t)}{\beta}$. To increase the occurrences of failures via importance sampling, the simulation workflow now accepts a point as a permanent failure event with the *failure biasing probability* P_{fb} instead of $\frac{\lambda(t)}{\beta}$. It also returns a *likelihood ratio* L for each event (instead of one in the original simulation as shown in Algorithm 1) to *unbiase* the estimate at the end of the simulation.

Both β and P_{fb} need to be properly configured. For β, it should be close to the average repair rate, while for P_{fb}, it is typically set as an intermediate value between 0 and 1 (e.g., 0.5) [12]. We evaluate the impact of β and P_{fb} in Section 5.4.

Integrating importance sampling into SIMEDC: We show how we leverage importance sampling (based on uniformization-based balanced failure biasing) to estimate the PDL in accelerated mode. A challenge here is that we need to address both permanent disk failures and permanent node failures in the estimation of the PDL. Thus, we address the following issues: (1) determination of the failure type (i.e., disk or node failures); (2) unbiasing the estimated PDL with the likelihood ratio; and (3) scheduling repair and failure events for different crashed disks or nodes. Currently, SIMEDC does not consider transient failures in importance sampling; how to apply importance sampling to transient failures is posed as future work.

Suppose that we have generated a failure event at time t. To determine whether the failure event is associated with a disk or a node, we define a *disk failure probability* P_{df}, which is calculated as the ratio of the sum of the failure rates of all available disks to the sum of the failure rates of all available disks and nodes. For example, if we choose the failure distributions shown in Table 1 (which we further elaborate in Section 5.1), the disk failure rate at time t is $\frac{1.12t^{0.12}}{(10years)^{1.12}}$, while the node failure rate remains at $\frac{1}{125\text{months}}$. With probability P_{df}, SIMEDC randomly chooses a disk from all available disks to fail; otherwise, it randomly chooses a node to fail. Thus, under importance sampling, a disk fails with probability $\frac{P_{fb}P_{df}}{A_d}$, where A_d is the number of available disks, while a node fails with probability $\frac{P_{fb}(1-P_{df})}{A_n}$, where A_n is the number of all available nodes.

Recall that without importance sampling, the actual failure probability of a subsystem s (a disk or a node) is $\frac{\lambda_s(t)}{\beta}$ under uniformization. Thus, we unbiase the likelihood ratio L by multiplying it with $\frac{\lambda_s(t)/\beta}{P_{fb}P_{df}/A_d}$ for a disk failure, or with $\frac{\lambda_s(t)/\beta}{P_{fb}(1-P_{df})/A_n}$ for a node failure.

To repair a failed subsystem, SIMEDC generates a repair event as in Algorithm 1, where the repair time is determined by the ratio of the amount of cross-rack bandwidth traffic to the available cross-rack bandwidth. SIMEDC runs a number of iterations until the RE is less than 20% (see Section 3.4).

Algorithm details: To incorporate importance sampling into SIMEDC, we mainly modify the GET_NEXT_EVENT function to determine how a failure event is generated. Algorithm 3 shows the details. If the system is normal, GET_NEXT_EVENT returns the first failure event to process; otherwise, if some subsystems fail, it activates importance sampling to accelerate the simulation. First, it draws a time t from an exponential distribution with the scale parameter β (Line 5). If t is larger than the earliest repair event time, the earliest repair event is returned to be the next event (Lines 6-7); otherwise, GET_NEXT_EVENT determines if the next event is a failure event based on the failure biasing probability P_{fb} (Line 8). If a failure event occurs, GET_NEXT_EVENT further checks if it belongs to a disk failure or a node failure, and accordingly updates L (which is defined as a global

Algorithm 3 Get_next_event under importance sampling

```
1: function GET_NEXT_EVENT
2:     if the whole system is normal then
3:         (time $t$, type $y$, subsystem $s$) $\leftarrow Q_f$.Pop
4:     else
5:         $t \leftarrow \exp(\beta) + t$
6:         if $t >$ time of next repair event then
7:             (time $t$, type $y$, subsystem $s$) $\leftarrow Q_r$.Pop
8:         else if $U(0,1) \leq P_{fb}$ then
9:             if $U(0,1) \leq P_{df}$ then
10:                $y \leftarrow$ permanent disk failure
11:                $s \leftarrow$ a randomly chosen disk
12:                $L \leftarrow L \times \frac{\lambda_s(t)/\beta}{P_{fb}P_{df}/A_d}$
13:            else
14:                $y \leftarrow$ permanent node failure
15:                $s \leftarrow$ a randomly chosen node
16:                $L \leftarrow L \times \frac{\lambda_s(t)/\beta}{P_{fb}(1-P_{df})/A_n}$
17:            end if
18:        else
19:            $y \leftarrow$ null
20:            $s \leftarrow$ null
21:            $L \leftarrow L \times \frac{1-\lambda(t)/\beta}{1-P_{fb}}$
22:        end if
23:    end if
24:    return (time $t$, type $y$, subsystem $s$)
25: end function
```

variable in SIMEDC) (Lines 8-22). Finally, it returns event time t, failure type y, and the subsystem s associated with the failure type if a failure event occurs.

5 SIMULATION RESULTS

In this section, we present the results of our reliability analysis based on SIMEDC.

5.1 Simulation Setup

Our simulation uses the following default settings unless otherwise specified.

Topology: We consider a data center with a total of 1,024 nodes that are evenly located in 32 racks (i.e., 32 nodes per rack). Each node is attached with one disk[1] of size 1 TiB, so the total storage capacity of the simulated data center is 1 PiB. We set the cross-rack bandwidth as 1 Gb/s, as obtained from Facebook's cluster measurements [30], and also set the chunk size as 256 MiB, as the default chunk size in Facebook's warehouses. We set the system mission time of the data center as 10 years [35]. While different erasure codes have different amounts of redundancy, we store the same number of data chunks, of a total size 0.5 PiB, for each erasure code setting.

Failure and repair models: Prior studies provide various statistical models for failure and repair patterns. Table 1 summarizes the default failure and repair models used in our simulation, and we justify our choices based on prior findings as follows.

- *Permanent disk failures:* The mean time of a permanent disk failure often ranges from few years (e.g., 4 years [30]) to

1. If a node is attached with multiple disks, we expect that the reliability of the data center will degrade as any permanent node failure is assumed to cause the data in all underlying disks to be permanently lost (see Section 2.1).

TABLE 1
Default failure and repair models.

Failure type	Time-to-failure	Repair time
Permanent disk failures	$\boldsymbol{W}$(1.12, 10 years, 0)	$\frac{\text{cross-rack repair traffic}}{\text{cross-rack bandwidth}}$
Permanent node failures	$\boldsymbol{Exp}(\frac{1}{125\text{ months}})$	
Transient node failures	$\boldsymbol{Exp}(\frac{1}{4\text{ months}})$	$\boldsymbol{Exp}(\frac{1}{0.25\text{ hours}})$
Transient rack failures	$\boldsymbol{Exp}(\frac{1}{10\text{ years}})$	$\boldsymbol{W}$(1, 24 hours, 10)
Permanent correlated failures	$\boldsymbol{Exp}(\frac{1}{1\text{ year}})$	$\boldsymbol{Exp}(\frac{1}{15\text{ hours}})$

$\boldsymbol{W}(\beta, \eta, \gamma)$ denotes a Weibull distribution with the shape parameter β, the characteristic life η, and the location parameter γ; $\boldsymbol{Exp}(\lambda)$ denotes an exponential distribution with the rate parameter λ.

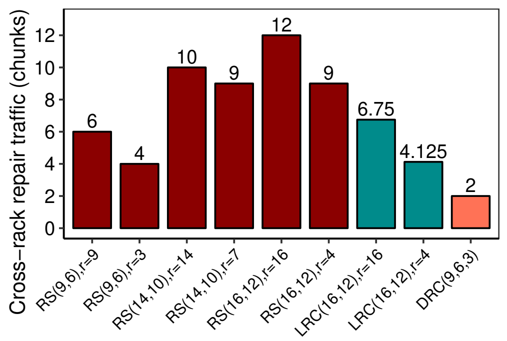


Fig. 4. Cross-rack repair traffic (in chunks) for different erasure code settings.

tens of years [6], [10], [35]. We model the time-to-failure as Weibull distributed with a characteristic life of 10 years. The repair time depends on the amount of cross-rack repair traffic and the cross-rack bandwidth (see Section 3.3).

- *Permanent node failures:* According to the statistics of Yahoo! cluster [34], about 0.8% of nodes permanently fail each month. Thus, we set the time-to-failure as exponentially distributed with mean 125 months. Like permanent disk failures, the repair time of a permanent node failure depends on the amount of cross-rack repair traffic and the cross-rack bandwidth.
- *Transient node failures:* A node temporarily fails once every 4 months, and the failure duration lasts no more than 15 minutes [10]. We set the time-to-failure and the repair time of a transient node failure as exponentially distributed with means 4 months and 15 minutes, respectively.
- *Transient rack failures:* We follow the same model in [35], in which the time-to-failure is exponentially distributed with mean 10 years [10], while the repair time is Weibull distributed with a characteristic life of 24 hours [7].
- *Permanent correlated failures:* The above failure types all belong to independent failures. We also consider a permanent correlated failure due to a power outage, which occurs once a year in production environments [34]. We set the time-to-failure as exponentially distributed with mean one year. We assume that a power outage affects a single rack and makes the rack temporarily unavailable until a power-on restart. We set the repair time of the power outage as exponentially distributed with mean 15 hours[2]. Furthermore, after the power-on restart, we permanently fail 1% of nodes in the rack, as in production environments [34]. We repair the permanent node failures as above.

Erasure codes: We compare RS codes, LRC, and DRC under flat and hierarchical chunk placements. We set the parameters n, k, l (for LRC only), and r (where $r = n$ and $r < n$ correspond to flat and hierarchical placements, respectively) based on production settings as follows.

- *RS(n, k):* We choose three settings of (n, k): RS(9,6) with $r=9$ and $r=3$, RS(14,10) with $r=14$ and $r=7$, and RS(16,12) with $r=16$ and $r=4$. Note that RS(9,6) is reportedly used by QFS [25], RS(14,10) is reportedly used by Facebook [23], and RS(16,12) correspond to the parameters of Windows Azure [18] (see below).
- *LRC(n, k, l):* We choose LRC(16,12,2), reportedly used by Windows Azure [18], with $r = 16$ and $r = 4$. In hierarchical placement, we place each local group of chunks in the fewest possible racks to minimize the cross-rack repair traffic in a single-chunk repair. In our case, we divide the $r = 4$ racks into two rack groups with two racks each, such that each of the $l = 2$ local groups of chunks (with six uncoded chunks and one local coded chunk) and a global coded chunk are placed in eight nodes of one rack group.
- *DRC(n, k, r):* We choose DRC(9,6,3), whose systematic code construction has been proposed [17].

2. We analyze failure records (see Section 5.5 for details) on node failures due to power outage or power spike, and find that the repair times range from 9 hours to 24 hours. Thus, we choose 15 hours as the average time for restoring a power outage.

Note that all the above erasure code settings have similar amounts of storage redundancy (i.e., n/k) between 1.33× and 1.5×. Figure 4 illustrates the cross-rack repair traffic (in unit of chunks) for a single-chunk repair of different erasure code settings; for LRC codes, we average the cross-rack repair traffic for each type of chunks (see Section 2.3).

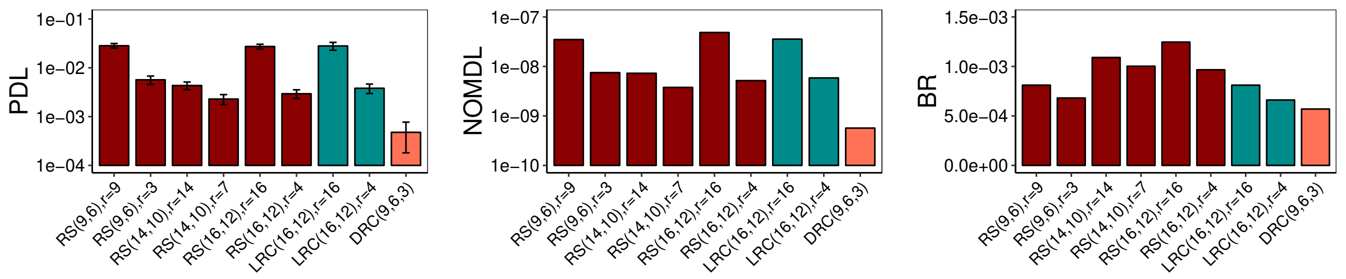
We present the results of PDL, NOMDL, and BR. For the PDL, we also show the relative error; for both PDL and NOMDL, we use the log scale for the y-axis. By default, we disable importance sampling unless specified otherwise.

5.2 Independent Failures

We first study the reliability of various erasure code settings under independent failures (i.e., the first four failures in Table 1), by disabling the permanent correlated failures.

Frequency of single-chunk repairs: We examine the repair events in our simulation, and find that over 99.5% of repairs are single-chunk repairs for all erasure code settings. Thus, the repair time mostly depends on the amount of cross-rack repair traffic of a single-chunk repair (as shown in Figure 4) of each erasure code setting.

Erasure codes in flat placement: Figure 5 shows the reliability results under independent failures only based on the default settings; in particular, the cross-rack bandwidth is 1 Gb/s. We first consider RS codes and LRC in flat placement (i.e., $r = n$). RS(14,10) has the lowest PDL and NOMDL among all RS codes, as it tolerates more failed chunks than RS(9,6) and has less repair traffic than RS(16,12). Note that LRC(16,12,2) has almost the same PDL as RS(16,12) even though it incurs less repair traffic, mainly because it is non-MDS and cannot tolerate all combinations of four failed chunks as RS(16,12). However, LRC(16,12,2) has less NOMDL



(a) PDL (b) NOMDL (c) BR

Fig. 5. Reliability under independent failures only, with the cross-rack bandwidth 1 Gb/s.

(by 26.5%) as it may have fewer failed chunks when data loss occurs.

Comparison of flat placement and hierarchical placement: We again use Figure 5 to compare flat placement (i.e., $r = n$) and hierarchical placement (i.e., $r < n$). Hierarchical placement generally achieves better reliability than flat placement for the same erasure code, mainly because of the reduction of cross-rack repair traffic. For example, compared to flat placement, hierarchical placement decreases the PDL of RS(9,6) by 80% and that of LRC(16,12,2) by 89%. In particular, DRC achieves the best reliability among all erasure code settings we consider; the relative error of PDL is high due to the small value of PDL (see Equation 1). We also observe that the BR closely matches the cross-rack repair traffic of each erasure code setting as shown in Figure 4; that is, the BR increases with the amount of cross-rack repair traffic.

Impact of cross-rack bandwidth: We also vary the cross-rack bandwidth (including 400 Mb/s, 2 Gb/s, 5 Gb/s, and 10 Gb/s) in our reliability evaluation; in the interest of space, we refer readers to our digital supplementary file for the results. To summarize, if the cross-rack bandwidth is 400 Mb/s, the erasure code settings under flat placement have PDL equal to (or nearly equal to) one (i.e., data loss always occurs), while DRC(9,6,3) has PDL equal to 1.26e-2. This shows the significance of minimizing the cross-rack repair traffic under limited cross-rack bandwidth. If the cross-rack bandwidth is 2 Gb/s or higher, the repair performance improves and hence the PDL significantly decreases, in which some erasure code settings do not observe any data loss. In our digital supplementary file, we also report the evaluation results for the cross-rack bandwidth of 5 Gb/s and 10 Gb/s for the cases of correlated failures (Section 5.3) and importance sampling (Section 5.4).

5.3 Correlated Failures

We now add permanent correlated failures to our simulation in addition to independent failures. Our investigation finds that over 99.3% of repairs are single-chunk repairs, so the repair time is still mainly determined by the amount of cross-rack repair traffic of a single-chunk repair. Figure 6 shows the results. For RS(14,10) and RS(16,12), they both incur high cross-rack repair traffic, so hierarchical placement can decrease their PDL values by reducing the cross-rack repair traffic. However, for RS(9,6) and LRC(16,12,2), although hierarchical placement reduces BR, it has worse PDL and NOMDL than flat placement as it sacrifices rack-level fault tolerance and becomes more vulnerable to correlated failures.

TABLE 2
Comparison of PDL (with the relative error) and running time per iteration with and without importance sampling.

Erasure codes	β	**PDL**$_{is}$	**PDL**$_{reg}$	**T**$_{is}$**(s)**	**T**$_{reg}$**(s)**
RS(9,6), $r = 9$	0.095	3.12e-2±18%	2.78e-2±18%	15.9	90.8
RS(9,6), $r = 3$	0.143	6.11e-3±13%	6.22e-3±22%	16.4	99.4
RS(14,10), $r = 14$	0.090	1.01e-2±10%	4.72e-3±23%	24.4	91.2
RS(14,10), $r = 7$	0.105	4.16e-3±16%	2.44e-3±28%	24.0	105.6
RS(16,12), $r = 16$	0.069	5.93e-2±4%	2.58e-2±19%	21.9	105.8
RS(16,12), $r = 4$	0.090	1.34e-2±9%	1.90e-3±32%	22.2	103.2
LRC(16,12), $r = 16$	0.094	3.69e-2±17%	3.27e-2±20%	20.0	197.2
LRC(16,12), $r = 4$	0.150	4.43e-3±15%	5.76e-3±23%	20.9	202.4
DRC(9,6,3)	0.280	2.41e-4±20%	2.50e-4±88%	17.7	110.4

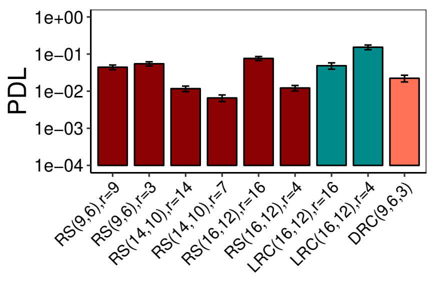
Note that DRC(9,6,3) still achieves higher reliability than RS(9,6) with $r = 3$.

5.4 Impact of Importance Sampling

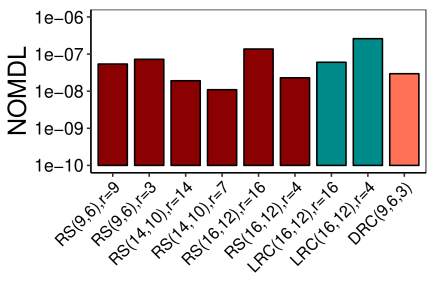
We now study the impact of importance sampling (based on uniformization-based balanced failure biasing) in terms of the accuracy and performance of SIMEDC. Based on previous studies [12], [20], [24], we configure β as the average repair rate for repairing a permanent failure (i.e., the ratio of the cross-rack bandwidth to the cross-rack repair traffic as shown in Table 1) and $P_{fb} = 0.5$. Here, we focus on the PDL analysis under independent permanent failures, as transient failures do not affect the PDL results. We also disable correlated failures as in Section 5.2.

Table 2 compares the results with importance sampling and the results of *regular simulation* (i.e., without importance sampling) with the cross-rack bandwidth of 1 Gb/s. We first compare their PDLs (denoted by PDL$_{is}$ and PDL$_{reg}$, respectively). We observe that while PDL$_{is}$ differs from PDL$_{reg}$ for each erasure code, the results of importance sampling still preserve the following two types of relative differences: (i) the relative differences across different erasure codes under the flat placement (i.e., different (n, k) for $n = r$) and (ii) the relative differences between the flat and hierarchical placements for the same erasure code (i.e., different r for the same (n, k)).

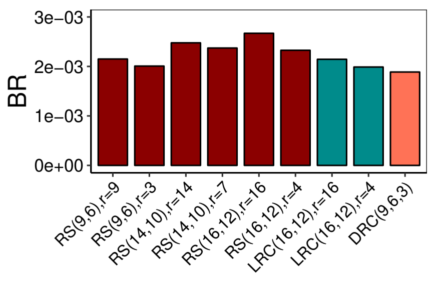
We next evaluate the running time per iteration of importance sampling and regular simulation (denoted by T$_{is}$ and T$_{reg}$, respectively). We observe that importance sampling reduces the running time by 73-90% since importance sampling increases the occurrences of the failure/repair events before a data loss occurs or the mission time is reached.



(a) PDL



(b) NOMDL



(c) BR

Fig. 6. Reliability under both independent and correlated failures.

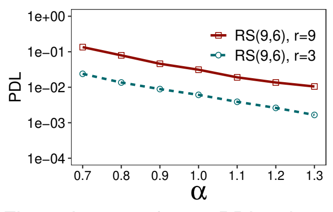


Fig. 7. Impact of β on PDL, where $\beta = \alpha R$.

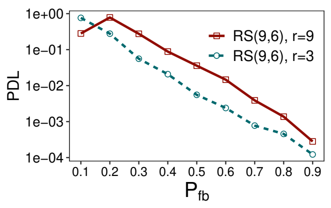


Fig. 8. Impact of P_{fb} on PDL.

Impact of β: We now evaluate the impact of different values of β. Here, we set $\beta = \alpha R$, where R denotes the average repair rate of a permanent failure and α is the weight that we vary in our evaluation (by default, we set $\alpha = 1$). Figure 7 presents the PDL of RS(9,6) with varying β based on different values of α. We observe that the PDL decreases with β, but remains at the same order of magnitude when α is between 0.9 and 1.2. This confirms the rationale of setting β to be close to the average repair rate. Note that the PDL of RS(9,6) with $r = 3$ (hierarchical placement) remains lower than with $r = 9$ (flat placement) for different values of β, which conforms to our analysis in regular simulation.

Impact of P_{fb}: We now study the PDL for different values of P_{fb}. Figure 8 shows the PDL of RS(9,6) versus P_{fb}. We observe that if P_{fb} is too low (e.g., $P_{fb} = 0.1$), the PDL is close to one, making our analysis inaccurate. On the other hand, if $P_{fb} > 0.1$, the PDL of RS(9,6) with $r = 3$ remains lower than with $r = 9$ due to the reduction of cross-rack repair traffic.

5.5 Trace Analysis

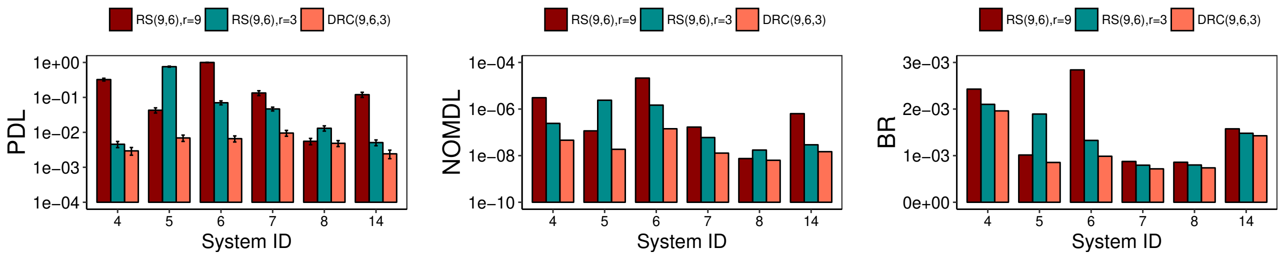
We now evaluate the reliability of different erasure code settings based on production traces of failure and repair events. We consider traces (downloadable from [1]) from high performance computing (HPC) environments reported by Schroeder *et al.* [31]. The traces span 22 HPC systems of one to 1,024 nodes each at Los Alamos National Laboratory. They contain failure records about node failures. Each record includes the time when the failure starts, the time when it is repaired, the root cause labeled by system operators, etc.

In our analysis, we focus on large-scale HPC systems with at least 128 nodes each and deploy them as hierarchical data centers. Thus, we select a total of 14 HPC systems, whose system IDs are 4-11 and 13-18 [31]. They have 128, 164, 256, 512, or 1024 nodes each, and we partition the nodes evenly into 16, 41, 32, 32, and 32 racks, respectively. We follow the default settings in Section 5.1 to configure each system. Note that the traces span less than the system mission time (10 years in our case). In our simulation, after a trace reaches the end, we replay it from the beginning to end, and repeat the replay process until the system mission time is reached.

We parse the failure records and categorize the failures based on their root cause labels. If the root causes are related to network slowdown, maintenance, or power outage (e.g., “Network”, “Console Network Device”, “Maintenance”, “Power Outage”, and “Power Spike”), we treat them as transient node failures and obtain their repair times directly from the failure records. If the root causes are related to disks (e.g., “Disk Drive”, “SCSI Controller”, “SAN Controller”), we treat them as permanent node failures (which also bring down the attached disks). We set the repair times based on the amount of cross-rack repair traffic and cross-rack bandwidth to reflect how much failed data needs to be repaired. For permanent disk failures and transient rack failures, we do not observe them in our traces, but we still generate them based on the models in Table 1. For permanent correlated failures, we do not specifically generate them, but we observe that a contiguous set of nodes fail within a short time in our traces (see discussion below).

We mainly compare RS(9,6) with $r = 9$ and $r = 3$, as well as DRC(9,6,3). As in our previous experiments that derive failure and repair events from statistical models, we find that single-chunk repairs still dominate and account for over 98.2% of all repairs. Also, we find that eight of the 14 systems (whose IDs are 9, 10, 11, 13 and 15-18) have almost zero values in all three metrics, so we only plot the results for the remaining six systems, as shown in Figure 9. For system IDs 4, 6, 7, and 14, we observe the same trends in as our previous experiments. That is, hierarchical placement is more reliable than flat placement, and DRC achieves the best reliability by minimizing the cross-rack repair traffic.

However, we find that for system ID 5, RS(9,6) under hierarchical placement has the worst reliability, while for system ID 8, it has the highest PDL and NOMDL. Our investigation finds that some contiguous nodes fail within a short time. For example, for system ID 5, we observe that nodes 16-19 in the same rack fail within 13 hours. For hierarchical placement, if three chunks of a stripe are stored in those failed nodes, then an additional failed chunk will lead to data loss before they are repaired. Flat placement is more robust against this type of contiguous node failures by storing only one chunk of a stripe in a distinct rack. Nevertheless, DRC(9,6,3) still achieves the best reliability among all three erasure code settings.



(a) PDL (b) NOMDL (c) BR

Fig. 9. Reliability under trace-driven failures. Although the sequence of events from traces is deterministic, we still observe relative errors in PDL, mainly because we generate permanent disk failures and transient rack failures from statistical models and the chunk positions vary across iterations.

5.6 Summary

We summarize the key findings of our simulation as follows.

- When there are independent failures only, hierarchical placement generally achieves better reliability than flat placement by reducing the cross-rack repair traffic. Among all erasure code settings, DRC achieves the best reliability. In particular, the BR increases with the amount of cross-rack repair traffic.
- The significance of reducing cross-rack repair traffic is more prominent in extreme scenarios (e.g., when the available cross-rack bandwidth is limited).
- When there are correlated failures, hierarchical placement may have higher PDL and NOMDL than flat placement as it tolerates fewer rack failures. Nevertheless, for erasure codes with high repair traffic (e.g., RS(14,10) and RS(16,12)), hierarchical placement still achieves better reliability.
- SIMEDC can accelerate simulations via importance sampling, while preserving the accuracy of reliability analysis.
- We make consistent observations for both statistically generated and trace-driven failure and repair events.

6 Related Work

We review related work on reliability studies of distributed storage systems, from modeling and simulation perspectives.

Modeling: Most reliability studies are based on Markov modeling, under the assumptions that both failure and repair times follow exponential distributions. Weatherspoon and Kubiatowicz [37] show via Markov modeling that erasure coding incurs significantly less bandwidth and storage overhead than replication for the same reliability. Rao *et al.* [26] model the redundancy within and across storage nodes. They show that the reliability heavily depends on the node repair time, which depends on the amount of data transferred for repair. Ford et al. [10] model stripe availability of Google storage subject to factors such as redundancy policies, recovery rates, and the presence of correlated failures. Some studies (e.g., [18], [30]) also analyze the reliability of new repair-friendly erasure code constructions based on Markov modeling. While the correctness of Markov modeling for reliability analysis is questionable [14], Iliadis *et al.* [19] justify the usefulness of Markov modeling and related non-Markov approaches for obtaining the MTTDL metrics.

In the context of chunk placement, Greenan *et al.* [13] use reliability modeling to determine the chunk placement of flat XOR-based erasure codes. Venkatesan *et al.* [36] analyze the reliability of erasure-coded storage with respect to chunk placement and repair rates. Hu *et al.* [17] present simplified Markov models to compare flat and hierarchical placements under special cases. Our work takes a simulation approach and complements existing modeling studies by considering more general and complicated failure/repair patterns.

Simulation: Several storage reliability simulators have been proposed in the literature. Greenan [12] presents the High-Fidelity Reliability Simulator (HFRS) for reliability simulation on disk arrays. We extend HFRS for data center environments. Silberstein *et al.* [35] develop a simulator to show the effectiveness of lazy repair (i.e., the repair of a stripe is deferred until its number of failed chunks exceeds a threshold) in distributed storage, but they do not consider hierarchical data centers. Fu *et al.* [11] conduct simulation analysis to study the reliability of primary storage when deduplication is deployed. Epstein *et al.* [9] combine simulation and combinatoric computations to estimate the durability of storage system, and take into account the available network bandwidth in the repair process. Hall [15] presents a simulator framework called CQSim-R, which evaluates the reliability in data center environments, and also studies the effects of chunk placement. Our work differs from previous simulators by specifically taking into account the impact of cross-rack repair traffic given the hierarchical nature of data centers. In addition, we consider more complicated failure patterns, including correlated failures and empirical failure traces; in CQSim-R [15], only independent disk-drive failures are considered.

7 Conclusions

We present SIMEDC, a discrete-event simulator that characterizes the reliability of erasure-coded data centers. SIMEDC specifically addresses the hierarchical nature of data centers and analyzes how various erasure code constructions and chunk placement schemes affect the overall storage reliability due to different amounts of cross-rack repair traffic. We demonstrate how SIMEDC can accelerate the simulation process via importance sampling. We present extensive reliability analysis results based on SIMEDC. In future work, we plan to extend SIMEDC to support different repair scheduling strategies, such as parallelization of a single-chunk repair operation [21], [22].

Acknowledgments

This work was supported in part by the Research Grants Council of Hong Kong (GRF 14216316 and CRF C7036-15G).

REFERENCES

[1] Failure Trace Archive. http://fta.scem.uws.edu.au/index.php?n=Main.Download.

[2] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. Vijaykumar. ShuffleWatcher: Shuffle-aware Scheduling in Multi-tenant MapReduce Clusters. In *Proc. of USENIX ATC*, 2014.

[3] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. In *Proc. of ACM SIGCOMM*, 2008.

[4] R. J. Chansler. Data Availability and Durability With the Hadoop Distributed File System. *The USENIX Magzine*, 37(1), 2012.

[5] M. Chowdhury, S. Kandula, and I. Stoica. Leveraging Endpoint Flexibility in Data-Intensive Clusters. In *Proc. of ACM SIGCOMM*, 2013.

[6] A. Cidon, R. Escriva, S. Katti, M. Rosenblum, and E. G. Sirer. Tiered Replication: A Cost-effective Alternative to Full Cluster Geo-replication. In *Proc. of USENIX ATC*, 2015.

[7] J. Dean. Designs, Lessons and Advice from Building Large Distributed Systems. http://www.cs.cornell.edu/projects/ladis2009/talks/dean-keynote-ladis2009.pdf.

[8] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. J. Wainwright, and K. Ramchandran. Network Coding for Distributed Storage Systems. *IEEE Trans. on Info. Theory*, 56(9):4539–4551, 2010.

[9] A. Epstein, E. K. Kolodner, and D. Sotnikov. Network Aware Reliability Analysis for Distributed Storage Systems. In *Proc. of IEEE SRDS*, 2016.

[10] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in Globally Distributed Storage Systems. In *Proc. of USENIX OSDI*, 2010.

[11] M. Fu, P. P. C. Lee, D. Feng, Z. Chen, and Y. Xiao. A Simulation Analysis of Reliability in Primary Storage Deduplication. In *Proc. of IEEE IISWC*, 2016.

[12] K. M. Greenan. Reliability and Power-efficiency in Erasure-coded Storage Systems. *UC Santa Cruz, Tech. Rep. UCSC–SSRC–09–08*, 2009.

[13] K. M. Greenan, E. L. Miller, and J. J. Wylie. Reliability of Flat XOR-based Erasure Codes on Heterogeneous Devices. In *Proc. of IEEE DSN*, 2008.

[14] K. M. Greenan, J. S. Plank, and J. J. Wylie. Mean Time to Meaningless: MTTDL, Markov Models, and Storage System Reliability. In *Proc. of USENIX HotStorage*, 2010.

[15] R. J. Hall. Tools for Predicting the Reliability of Large-Scale Storage Systems. *ACM Trans. on Storage*, 12(4):24, 2016.

[16] Y. Hu, P. P. C. Lee, and X. Zhang. Double Regenerating Codes for Hierarchical Data Centers. In *Proc. of IEEE ISIT*, 2016.

[17] Y. Hu, X. Li, M. Zhang, P. P. C. Lee, X. Zhang, P. Zhou, and D. Feng. Optimal Repair Layering for Erasure-Coded Data Centers: From Theory to Practice. *ACM Trans. on Storage*, 13(4), 2017.

[18] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin. Erasure Coding in Windows Azure Storage. In *Proc. of USENIX ATC*, 2012.

[19] I. Iliadis and V. Venkatesan. Rebuttal to ”Beyond MTTDL: A Closed-Form RAID-6 Reliability Equation”. *ACM Trans. on Storage*, 11(2):9, 2015.

[20] P. W. Lewis and G. S. Shedler. Simulation of Nonhomogeneous Poisson Processes by Thinning. *Naval Research Logistics Quarterly*, 26(3):403–413, 1979.

[21] R. Li, X. Li, P. P. C. Lee, and Q. Huang. Repair Pipelining for Erasure-Coded Storage. In *Proc. of USENIX ATC*, 2017.

[22] S. Mitra, R. Panta, M.-R. Ra, and S. Bagchi. Partial-Parallel-Repair (PPR): A Distributed Technique for Repairing Erasure Coded Storage. In *Proc. of ACM EuroSys*, 2016.

[23] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, and K. Sanjeev. f4: Facebooks Warm Blob Storage System. In *Proc. of USENIX OSDI*, 2014.

[24] V. F. Nicola, P. Heidelberger, and P. Shahabuddin. *Uniformization and Exponential Transformation: Techniques for Fast Simulation of Highly Dependable non-Markovian Systems*. IBM Research Division, TJ Watson Research Center, 1992.

[25] M. Ovsiannikov, S. Rus, D. Reeves, P. Sutter, S. Rao, and J. Kelly. The Quantcast File System. In *Proc. of VLDB Endowment*, 2013.

[26] K. Rao, J. L. Hafner, and R. A. Golding. Reliability for Networked Storage Nodes. *IEEE Trans. on Dependable and Secure Computing*, 8(3):404–418, May/June 2011.

[27] K. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran. A Solution to the Network Challenges of Data Recovery in Erasure-coded Distributed Storage Systems: A Study on the Facebook Warehouse Cluster. In *Proc. of USENIX HotStorage*, 2013.

[28] K. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran. A Hitchhiker’s Guide to Fast and Efficient Data Reconstruction in Erasure-coded Data Centers. In *Proc. of ACM SIGCOMM*, 2014.

[29] I. S. Reed and G. Solomon. Polynomial Codes over Certain Finite Fields. *Journal of the Society for Industrial and AppliedMathematics*, 8(2):300–304, 1960.

[30] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur. Xoring Elephants: Novel Erasure Codes for Big Data. In *Proc. of VLDB Endowment*, 2013.

[31] B. Schroeder and G. Gibson. A Large-Scale Study of Failures in High-Performance Computing Systems. *IEEE Trans. on Dependable and Secure Computing*, 7(4):337–350, 2010.

[32] 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 *Proc. of USENIX FAST*, 2007.

[33] Z. Shen, J. Shu, and P. P. C. Lee. Reconsidering Single Failure Recovery in Clustered File Systems. In *Proc. of IEEE DSN*, 2016.

[34] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *Proc. of IEEE MSST*, 2010.

[35] M. Silberstein, L. Ganesh, Y. Wang, L. Alvisi, and M. Dahlin. Lazy Means Smart: Reducing Repair Bandwidth Costs in Erasure-coded Distributed Storage. In *Proc. of ACM SYSTOR*, 2014.

[36] V. Venkatesan and I. Iliadis. Effect of Codeword Placement on the Reliability of Erasure Coded Data Storage Systems. In *Proc. of QEST*, 2013.

[37] H. Weatherspoon and J. D. Kubiatowicz. Erasure Coding Vs. Replication: A Quantitative Comparison. In *Proc. of IPTPS*, 2002.

[38] M. Zhang, S. Han, and P. P. C. Lee. A Simulation Analysis of Reliability in Erasure-Coded Data Centers. In *Proc. of IEEE SRDS*, 2017.

Mi Zhang received the B.Eng. degree in Software Engineering from Shandong University in 2014. She is now pursuing her Ph.D. degree in Computer Science and Engineering at the Chinese University of Hong Kong. Her research interests include distributed systems and storage reliability.

Shujie Han is currently a Ph.D. student in the Department of Computer Science and Engineering at the Chinese University of Hong Kong. Her research interests include data deduplication, reliability, etc.

Patrick P. C. Lee received the B.Eng. degree (first-class honors) in Information Engineering from the Chinese University of Hong Kong in 2001, the M.Phil. degree in Computer Science and Engineering from the Chinese University of Hong Kong in 2003, and the Ph.D. degree in Computer Science from Columbia University in 2008. He is now an Associate Professor of the Department of Computer Science and Engineering at the Chinese University of Hong Kong. His research interests are in storage systems, distributed systems and networks, and cloud computing.