Extreme Protection against Data Loss with Single-Overlap Declustered Parity

Huan Ke, Haryadi S. Gunawi
University of Chicago
Chicago, USA
{huanke,haryadi}@uchicago.edu

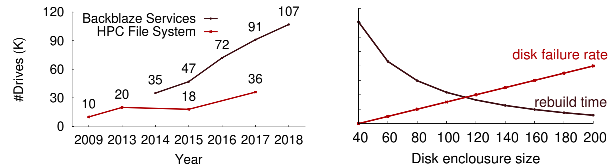
David Bonnie, Nathan DeBardeleben, Michael Grosskopf,
Terry Grové, Dominic Manno, Elisabeth Moore, Brad Settlemyer
Los Alamos National Laboratory
Los Alamos, NM, USA
{dbonnie,ndebard,gross,tagrove,dmanno,lissa,bws}@lanl.gov

***Abstract*—Massive storage systems composed of tens of thousands of disks are increasingly common in high-performance computing data centers. With such an enormous number of components integrated within the storage system the probability for correlated failures across a large number of components becomes a critical concern in preventing data loss. In this paper we reconsider the efficiency of traditional declustered parity data protection schemes in the presence of correlated failures. To better protect against correlated failures we introduce Single-Overlap Declustered Parity (SODP), a novel declustered parity design that tolerates more disk failures than traditional declustered parity. We then introduce CoFaCTOR, a tool for exploring operational reliability in the presence of many types of correlated failures. By seeding CoFaCTOR with real failure traces from LANL's data center we are able to create a failure model that accurately describes the existing file system's failure model and can use that model to generate failure data for hypothetical system designs. Our evaluation using CoFaCTOR traces shows that when compared to the state of the art our SODP-based placement algorithms can achieve a 30x improvement in the probability of data loss during failure bursts and achieves similar data protection using only half as much parity overhead.**

I. Introduction

Massive storage systems, such as those used for cloud archival services [1], [2] and the file systems at high-performance computing (HPC) data centers [3], [4] provide critical data services to users and are increasingly relied on to never lose data. At the same time denser and larger disk drives make these storage systems at greater risk for catastrophic failures and data loss [5]. These systems rely heavily on RAID technology using declustered parity, to provide fault tolerance and prevent the loss of valuable data – however, declustered parity schemes were not designed to tolerate large numbers of failures in short windows of time.

Figure 1(a) shows how storage systems at both cloud archival services and HPC data centers at national laboratories have grown to include tens and even hundreds of thousands of disk drives. At such large drive counts the probability of failure bursts and the risk of data loss due to a failure burst are both increasing. Similarly in Figure 1(b) we see that as the number of drives within a parity-protected enclosure increases, the likelihood of drive failures increases at a linear rate while improvements in rebuild time decrease as the amount of rebuild data for a single disk failure remains fixed. With modern disk enclosures routinely incorporating 84 or 106 disk drives [6], [7] and file systems commonly spanning multiple large disk enclosures it becomes imperative to develop new methods for preventing data loss.



(a) Storage System Disks over Time (b) Disk Fault Tolerance

Fig. 1: Figure 1a shows the growth in the number of drives used at Backblaze (a cloud service providing data archiving) and the parallel file systems deployed at LANL, ORNL, and LLNL HPC data centers. Figure 1b shows how the rate of disk failures (5% failure rate) increases linearly as disks are added to the enclosure while the time to rebuild data decreases more slowly (with a 50MB/s disk rebuild rate) because the amount of data to rebuild for a single failure remains fixed (e.g. 8TB).

At the same time we see an additional change in the nature of storage system component failures. Modern data protection schemes and the mean time to data loss calculations that motivate them both assume that drive failures are independent and identically distributed [8], [9]. However, recent studies describing state of the art data protection in large-scale cloud data centers have instead explored the likelihood of data loss during correlated failure bursts [10]–[12]. Similarly, in this paper we present data from Los Alamos National Laboratory's (LANL) file system attached to the Trinity supercomputer [13] that indicates that correlated failures are common within that storage system as well.

Based on the emergence of extremely large disk enclosures and a new requirement to tolerate bursts of correlated failures we have developed a new data protection scheme called Single-Overlap Declustered Parity (SODP). With SODP we have created a new set of data placement schemes for declustered parity data protection that focuses on maximizing the number of disk failures tolerated while also minimizing disk rebuild time. To make SODP generally useful we provide algorithms for creating SODP data placements for varying numbers of data blocks, parity blocks, and numbers of disks.

In order to evaluate SODP we developed a scheme for generating large volumes of trace data using a smaller set

of realistic failure traces. This tool, called the Correlated Failure Consultation Tool for Operational Reliability, or simply CoFaCTOR, develops an accelerated failure time model using regression statistics that enables the generation of large numbers of realistic failure traces. CoFaCTOR also enables the generation of traces that alter physical parameters, like the number of disks per enclosure, to test alternative designs.

By generating thousands of realistic failure traces we are able to leverage our event-driven simulation package, SOL-Sim, for modeling the failure, rebuild and replacement of drives within large-scale storage systems. Our simulation analysis enables us to determine that data placement algorithms based on the SODP principles can dramatically reduce the probability of data loss in the face of correlated failures.

The remainder of this paper is structured as follows: in Section II we review declustered parity and describe the progression of the state of the art for data protection, in Section III we present a description of single-overlap declustered parity including the algorithms for generating this data placement scheme and an analysis of how data protection is improved with our scheme, in Section IV we introduce the SOL-Sim design and in Section V we describe our methodology for evaluating single-overlap declustered parity using a combination of real data, realistic traces, and simulation and then describe how our schemes reduce the probability of data loss compared with current state of the art parity schemes, and in Section VI we present our study's conclusions.

II. Related Work

In this section, we review existing declustered parity approaches, and then discuss future directions for declustered parity data layouts. In 1990, Muntz and Lui [23] first introduced and analytically modeled declustered parity, but left the data placement decisions as an open problem. To address data placement, Holland and Gibson [14] implemented parity declustering based on Balanced Incomplete Block Designs (BIBD). As part of this work they identified six criteria for ideal declustered layouts:

- Single failure correcting, no two units of the same stripe are mapped to the same disk.
- Distributed reconstruction, when a disk fails, the reconstruction workload is evenly distributed across the surviving disks.
- Distributed parity, all disks have the same number of parity units.
- Efficient mapping, the mapping from client data to disk is implementable with low time and space requirements.
- Large write optimization, each parity stripe is aligned across the disks such that a stripe can be written without pre-reading the prior contents of any disk.
- Maximal parallelism, a read of n continuous data units induces parallel access from n disks.

BIBD($v, b, r, k,$) is a collection of b subsets of k elements over a set of v distinct objects, where each object appears in r subsets and each pair of two objects appears in subsets. When BIBD is applied in declustered parity, the essence is to find a data mapping to distribute parity stripes of size k over v disks (e.g., declustered layout), where each disk appears in r disk subsets and each pair of two disks appears in disk subsets. Note that the complete block design consists of $\binom{v}{k}$ subsets, where each object appears in exactly $\binom{v\;1}{k\;1}$ of the subsets.

DATUM [15] improves the data mapping by directly computing disks and offsets without using BIBD table lookup. The key idea is to utilize the complete block design, with a particular ordering of the $\binom{v}{k}$ disk subsets, and compute the disks and offsets through the orderings. One drawback of DATUM is the complete block designs were originally too large to be usable.

GridRAID [16] is the declustered parity scheme that used on LANL's Trinity file system. The basic idea is to divide the stripe data into tiles, each of which is a group of stripes across the disk array. In each tile, it does the data permutation to make the data, parity and spare space spread. With multiple tiles, it benefits from the distributed reconstruction workload during the recovery.

PRIME [17] is designed for prime values of v to approach the ideal declustered layout by slightly relaxing the maximal parallelism property. Like BIBD, PRIME constructs the declustered layout only for a limited set of configurations.

RELPR [17] is similar to PRIME, but it deviates from the ideal in two ways: maximal parallelism and distributed reconstruction. However, RELRP is applicable to arbitrarily configured disk array size v to achieve approximately balanced declustered layout via on-demand calculation.

PDDL and **dRAID** [18] declusters the layout by permuting the disks to spread the parity, spare, and client data units throughput the disk array. Obtaining satisfactory base disk permutations is a challenge which makes PDDL only applicable to limited configurations. [19] extended work based on PDDL that is designed for use within the Zettabyte File System (ZFS) [24]. To simplify the generation of base permutation, dRAID randomly generates multiple base permutations.

RAID+ [20] enables RAID construction over large disk enclosure to spread reconstruction workload in a balanced way. RAID+ utilizes the Latin squares to construct a declustered layout. The only problem is that the number of v-order mutually orthogonal Latin squares (MOLS) for general v is still an open problem, it's known to exist when v is a power of a prime number, which is exactly the same as BIBD.

D^3 [22] focuses on the data distribution in large-scale distributed storage system. Similar to declustered layout, D^3 designs a layout with orthogonal array to uniformly spread data and parity units across servers in the system.

OI-RAID [21] is a two-layer encoding architecture and uses BIBD in the outer layer to achieve a balanced data layout. It spreads the data and parity across BIBD groups to enable group fault tolerance, which implies at least three arbitrary disk failures.

Table I shows how the above schemes satisfy the six original declustered layout criteria. As we can see, all these existing works violate the properties of ideal data layout to

Schemes	Single Failure Correcting	Distributed Reconstruction	Distributed Parity	Efficient Mapping	Large Write Optimization	Maximal Parallelism	Configuration Limitation	#Fault Tolerance
BIBD [14]	✓	✓	✓		✓	M	✓	1
DATUM [15]	✓	✓	✓	✓	✓	M		m
GridRAID [16]	✓			✓	✓	L		m
PRIME [17]	✓	✓	✓	✓	✓	H	✓	m
REPLR [17]	✓		✓	✓	✓	M		m
PDDL [18]	✓	✓	✓	✓	✓	M	✓	m
dRAID [19]	✓			✓	✓	L		m
RAID+ [20]	✓	✓	✓	✓	✓	H	✓	m
OI-RAID [21]	✓	✓	✓		✓	M	✓	⩾3
D^3 [22]	✓	✓	✓	✓	✓	M	✓	m
SODP (ours)	✓	✓	✓	✓	✓	L		⩾m

TABLE I: Comparison of declustered layouts. H represents highly approaching maximal parallelism property, and M and L means medium and low, respectively. m is the maximal disk failures that be tolerated in the declustered layout.

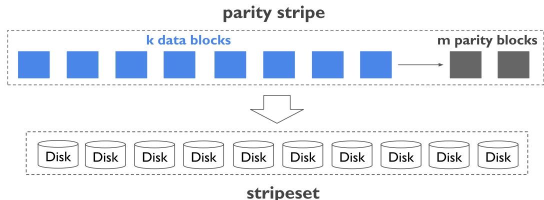


Fig. 2: Data organized as a parity stripe that will be distributed over a set of disks. A stripeset then is the specific disks selected for a one-to-one mapping with the data and parity blocks. As long as $k + m$ remains fixed, stripesets are independent of the parity stripe parameters.

some extent. Difficulties balancing these criteria lead many modern software based declustered parity schemes to use approximately balanced designs [25]. Finally, while many of these schemes have been extended to tolerate m failures our scheme, SODP, has been explicitly designed to tolerate greater than m failures.

III. Single-Overlap Declustered Parity

Prior work on parity declustering has often relied on known BIBD designs to construct perfectly balanced data layouts that attempt to maximize the six factors shown in Table I. However, as conceived, the six criteria do not seek to emphasize data survivability. To that end, we have identified two additional principles that emphasize data survivability during frequent failures:

> Maximizing the number of simultaneous disk failures tolerated without increasing parity overhead, and
> Minimizing disk rebuild time by balancing parity stripes across all disks.

In traditional declustered parity, data is encoded into k data blocks and m parity blocks with the $k + m$ blocks forming a **parity stripe**. In practice it is common to use Reed-Soloman codes to construct parity blocks and the notation for a parity scheme is typically shortened to RS(k,m). To satisfy the single failure correcting property, the parity stripe is stored onto a set of $k + m$ disks. In order to tolerate more than m disk failures our techniques require additional care in selecting how parity stripes are mapped onto disks, and thus we introduce the term **stripeset** to describe a set of disks onto which parity stripes

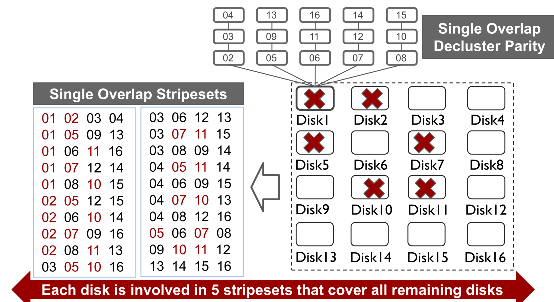


Fig. 3: A table of the full set of 4-disk single overlap stripesets chosen from a population of 16 total disks. With an RS(2,2) coding we can see that 6 simultaneous drive failures can be tolerated without data loss. The number of failures tolerated within a SODP layout depends on the parity scheme selected.

are mapped. Figure 2 shows an example of a parity stripe and stripeset. If more than m disk failures occur simultaneously within a single stripeset then the data in this stripeset is lost. In conventional RAID all stripes are located in a single stripeset. With *complete* parity declustering every possible permutation of disks exists as a valid stripeset.

Single-overlap declustered parity, or SODP, is a declustered layout scheme that ensures at most one overlapping disk between any two stripesets. This maximizes the number of disk failures that can be tolerated in a parity scheme with full declustering and maximizes the number of disks participating in a disk rebuild following a disk failure. Figure 3 illustrates the layout of the SODP design across 16 disks with RS(2, 2) encoding. As shown, only 20 total stripesets are required to construct a fully declustered layout, where every disk participates in a stripeset with every other disk. To provide an example, Disk1 participates in 5 stripesets, but none of the other disks appear more than once in those same stripesets. If Disk1 fails, the remaining 15 disks can be used for recovery, which provides the same rebuild performance as traditional parity declustering. This is true for all 16 disks. Furthermore, one can see that disks 1, 2, 5, 7, 10 and 11 may fail simultaneously and there is no stripeset experiencing 3 failures, and thus no data is lost. Therefore, rather than tolerating only 2 failures, the SODP layout can tolerate 6 failures without experiencing data loss.

A. Generating Single-Overlapping Stripesets

We introduce a stripeset construction algorithm we call Optimal SODP, or O-SODP, that uses matrix manipulation to minimize the number of stripesets. Before presenting the full O-SODP algorithm, we walk through the construction of single overlap stripesets using the above example, where each disk participates in 5 stripesets. First, the 16 disks are organized into a 4x4 disk matrix with rows a, b, c, d and columns $1, 2, 3, 4$ as shown in Figure 4. There are three steps to construct the single overlap stripesets:

1) Generate Row-based Stripesets: Each row (e.g., a, b, c, or d) consists of 4 disks, which form a row-based stripeset.

2) Generate Column-based Stripesets: Each column (e.g., 1, 2, 3, or 4) also consists of 4 disks, which construct a column-based stripeset.

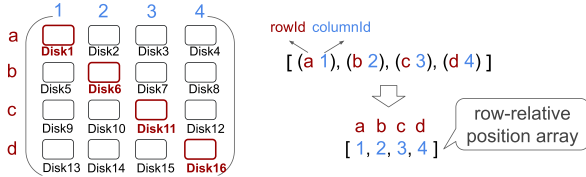


Fig. 4: Disk matrix and row-relative position array.

3) Generate Row-column Stripesets: The key idea of row-column stripesets is to choose 4 disks from different rows and columns. As shown in Figure 4, the simplest example is to choose disks on the diagonal, whose positions are denoted as [(a 1), (b 2), (c 3), (d 4)]. We simplify this notation into a *row-relative* position array $[1, 2, 3, 4]$, which represents the row-column stripeset [Disk1, Disk6, Disk11, Disk16].

To generate the remaining row-column stripesets while maintaining a balanced declustered layout we define a new algorithm called **shuffle permutation**. The objective is to swap all possible two position pairs in the diagonal row-relative position array to create new position arrays. As shown in Figure 5, if we first swap the position pair $(1, 2)$, it is obvious the next swap should be the pair $(3, 4)$, which generates a new *row-relative* position array $[2, 1, 4, 3]$ that denotes the row-column stripeset [Disk2, Disk5, Disk12, Disk15]. In this extremely simple example we produced three new row-relative position arrays which were permutation shuffled from the initial position array $[1, 2, 3, 4]$. After completing shuffle permutation, the resultant 4 position arrays form a position matrix, which has a unique value for each column. Therefore, the position matrix is able to generate 4 non-overlapping row-column stripesets which cover the entire disk matrix. Algorithm 1 shows the pseudocode for implementing the shuffle permutation algorithm.

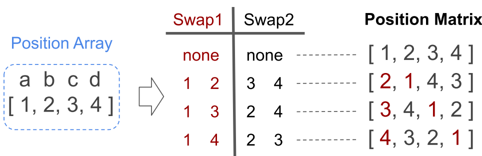


Fig. 5: Shuffle permutation by swapping all possible position pairs in the initial position array

Algorithm 1: Shuffle Permutations

```
Input: initialPosition ← [1,2,..., c], c columns
Output: P={P_1, P_2, ...}, shuffled position arrays
function CREATESHUFFLEARRAYS(B, N, k + m)
P = {}
for i = 1: c-1 do
    for j = i+1 : c do
        temp = initialPosition
        empty temp[i] and temp[j]
        P' = createShuffleArrays(temp)
        for k = 1: length(P') do
            tmp = P'_k
            tmp.insert(j) at i_th position
            tmp.insert(i) at j_th position
            P.add(tmp)
        end
    end
end
return P
end function
```

To generate the additional row-column stripesets containing at most one overlapping disk per stripeset, fix one position in the row (e.g., a) and rotate the other three positions of that row (e.g., b, c, d) as shown in Figure 6. A single rotation of the position array $[1, 2, 3, 4]$ leads to a new position array $[1, 3, 4, 2]$. By applying a single rotation to the other position arrays in the matrix, a new position matrix is generated. This newly formed position matrix corresponds to 4 non-overlapped row-column stripesets. The new stripesets are single overlapping with the position matrix from which they were derived. Furthermore, rotating the position array $[1, 2, 3, 4]$ twice leads to another new position array $[1, 4, 2, 3]$. Correspondingly, another new position matrix is formed to generate another 4 new row-column stripesets, all of which satisfy the single overlap property. The process will continue until rotation isn't possible anymore. Therefore, 3 position matrices are available from row-column stripeset generation, or alternatively each disk is included in three row-column stripesets.

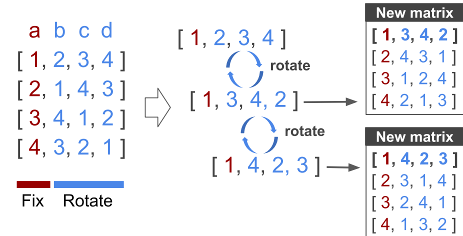


Fig. 6: Fix one position and rotate the remaining positions to generate the new position matrices.

To conclude this example, combining all row-based, column-based and row-column stripesets based on shuffle permutation and rotation, the above example generates 20 stripesets in total across 16 disks with each disk included in exactly 5 stripesets.

Challenges: When the size of a stripeset is large or an odd number, how do we do pair-wise swap in the permutation shuffle? Figure 7 illustrates the case of the stripeset size being 7, which leads to the initial diagonal position array $[1;2;3;4;5;6;7]$ with corresponding rows $a;b;c;d;e;f;g$. To generate a position matrix, we will swap 1 and 2 in the second position array, 1 and 3 in the third position array and so on. Unlike the previous 4-column case where the remaining pair-wise swap is obvious, our new 7-column case leaves the remaining swaps with $\frac{\binom{5}{2}\binom{3}{2}\binom{1}{1}}{2!}$ possibilities.

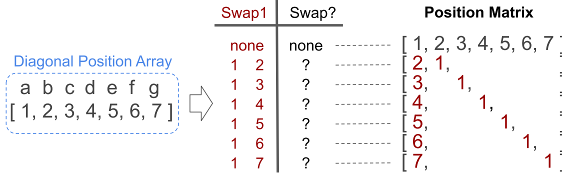


Fig. 7: Position array of size 7 and corresponding position matrix.

To solve the above challenge, we introduce the concept of **rotate distance**, which indicates the clockwise distance between any two positions in the rotate space of the position array. For example, the rotate distance from 3 to 6 is two, because it has to walk through 4 and 5. To satisfy the SODP property, we should guarantee the following constraint:

Constraint #1: Rotate distance before and after swapping cannot be equal.

With the same rotate distance, the new position array will eventually overlap multiple positions with the diagonal position array. As shown in Figure 8, if we swap the position pair $(3;6)$ in the second position array, the rotate distance from 3 to 6 is still two (e.g., walk through X and 1), which is equal to the previous rotate distance. As a result, after rotating the second position array 3 times, it will double overlap with the diagonal position array. To prevent this from occurring, we identify the following property to satisfy constraint #1 for any single position pair $(a;b)$.

$$d_r(a \to b) \neq d_r(b \to a)$$

where $d_r(a \to b)$ represents the rotate distance from a to b. In the example, this will prevent the swapping of position pairs $(3;6)$ and also $(4;7)$ in the second position array.

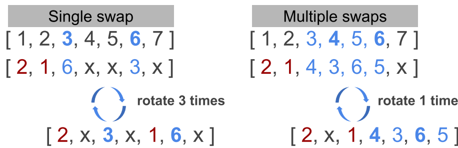


Fig. 8: Properties for single and multiple position pairs swapping in permutation shuffle.

Next, if we try to swap both $(3;4)$ and $(5;6)$ at the same time, the rotate distances $d_r(3 \to 5)$ and $d_r(4 \to 6)$ for the second position array are equivalent to those in the diagonal position array. This means if we were to then rotate the already swapped second position array 1 time, it would still cause an overlap (e.g., 4 and 6) with the diagonal position array. As we can see, both of the single position pair swaps are feasible, but swapping them together creates a conflict. To avoid the multiple overlaps resulting from multiple position pair swaps, we have to guarantee any two pairs $(a_1;b_1)$ and $(a_2;b_2)$ meet the following requirement to satisfy constraint #1.

$$d_r(a_1 \to b_1) \neq d_r(a_2 \to b_2)$$

This prevents the swapping of position pairs with the same rotate distance in the diagonal position array. For example, if we swap the position pair $(3;4)$, it will exclude other position pairs $(5;6)$ and $(6;7)$. The only feasible additional swap is the pair $(5;7)$ and then the remaining position 6 is left untouched. The resultant second position array is $[2;1;4;3;7;6;5]$, which will not overlap more than one position with the diagonal position array regardless of how many rotations are applied.

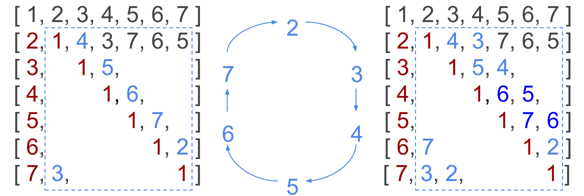


Fig. 9: Auto-generation of other position arrays with a given second position array.

Now we need to prevent multiple overlaps between the second and subsequent position arrays. The most straightforward way is to avoid swapping the same position pairs. Another approach, utilizing the given second position array, would be to add one along the diagonal based on the circle shown in Figure 9. As you can see, the positional elements inserted alongside the diagonal 1s are $4;5;6;7;2;3$. By applying the same principle, the next set of additional positional elements are $3;4;5;6;7;2$. At this point, one can notice the repeating $6;5$ and $7;6$, which have already occurred in the second position array. This materializes because in the second position array $d_r(4 \to 3)$ is equal to $d_r(6 \to 5)$, and by adding 1 both $6;5$ and $7;6$ appear again thus causing a double overlap in the given second position array. To guarantee the derived arrays will never violate the SODP constraint with the second position array, we identify a second constraint:

Constraint #2: Rotate distances in the second position array must be distinct.

This means if we have the successive $4;3$ in the second position array, it is not allowed to include successive $7;6$ or $6;5$. Otherwise, other derived position arrays will experience multiple overlaps with the second position array.

By combining the two constraints above, we are able to create a feasible second position array and a corresponding position matrix. Note that sometimes, a perfectly balanced declustered layout based on our position matrix with the required parameters cannot be found, we publish our feasible position matrices in [26].

Theoretical Analysis To demonstrate that O-SODP minimizes the number of stripesets, S, we assume a disk array

of size N, where each stripeset consists of $k + m$ disks. To count the disk pairs $(i; j)$, we have

$$S * \frac{(k+m)(k+m-1)}{2} \geq \frac{N(N-1)}{2}$$

which guarantees the disk pairs in stripesets cover all disks pairs in the N-disk array. The size of S can be formulated as:

$$S \geq \frac{N(N-1)}{(k+m)(k+m-1)}$$

To count the number of pairs $(s; d)$ where s is a stripeset and d is a disk in the stripeset, we have the following equation:

$$S * (k+m) = N * r$$

Here r is the number of stripesets per disk. To count the triples $(s; d_1; d_2)$ where d_1 and d_2 are distinct disks and s is a stripeset that contains both, we have the following equation:

$$1 * (N-1) = r * (k+m-1)$$

where O-SODP makes any pair of disks (e.g., d_1 and d_2) appear in one stripeset. By combining the two equations, the size of S equals $\frac{N(N-1)}{(k+m)(k+m-1)}$, which is the minimum.

Figure 10 compares the total number of stripesets using O-SODP with the configurations identified in prior BIBD literature [14]. Additionally, we compare the number of stripesets per disk using O-SODP, being that the number of stripesets per disk directly reflects the rebuild performance. To be specific, the number of surviving disks participating in single disk rebuild is:

$$min\{stripesets\text{-}per\text{-}disk * (k+m-1); N-1\}$$

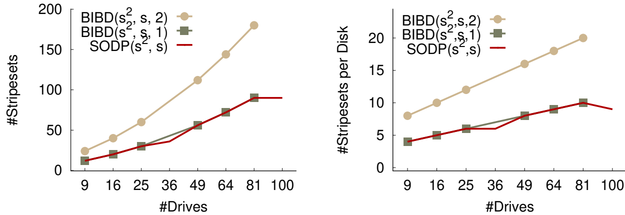


Fig. 10: Total number of stripesets and number of stripesets per disk for SODP and defined BIBD configurations.

We see that in general our O-SODP algorithm is able to match the BIBD performance with = 1 while the dips show the G-SODP (see Section III-B) results for configurations not having a known BIBD configuration. We also include the BIBD stripeset counts for the same configuration with = 2 to demonstrate the degree to which higher generate additional stripesets which do not further improve rebuild performance but do reduce the total number of disk failures that can be tolerated. If we consider a stripeset as a failure domain we see that these BIBD designs have a greater number of failure domains with a lower degree of fault tolerance. However, O-SODP is not guaranteed to generate a set of single-overlap stripesets for all configurations (even when the stripeset size is smaller than the square root of the number of disks).

B. Greedy SODP

For arbitrary numbers of disks and arbitrary numbers of data blocks (k) and parity blocks (m) we designed the Greedy SODP algorithm, G-SODP, to achieve *nearly* single overlap declustering. Put simply, G-SODP sacrifices a small amount of rebuild performance to gain a modest improvement in disk failure tolerance. As we will see later in Section V this tradeoff turns out to be surprisingly effective when we evaluate the probability of data loss under failure bursts. In other words, G-SODP achieves a result very similar to that of O-SODP by slightly reducing the rebuild performance, which in turn can tolerate more disk failures.

The basic idea of G-SODP is to create one or more base stripesets and derive the i_{th} stripeset by adding i mod N. Figure 11 illustrates how to construct 2 + 2 stripesets within 16 disks. G-SODP uses $[1; 3; 6; 7]$ as the base stripeset and add i to the disk obtained from the base stripeset, which is able to generate 15 derived stripesets. Utilizing this process, G-SODP guarantees each disk participates in an equal number of stripesets (e.g., 4) to achieve a balanced declustered layout comparable to O-SODP.

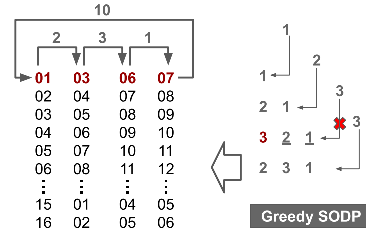


Fig. 11: Greedy SODP

Challenges: The fundamental challenge of G-SODP is to obtain base stripesets, which aim to construct distinct distances between any two elements inside the given base stripeset. In Figure 11, the base stripeset $[1; 3; 6; 7]$ is comprised of distinct distances 2, 3, 1, and 10. As we can see, any one, two or three accumulative distances are impossible to be equal to any other existing distance, which guarantees adding any i will not yield more than one overlapping disk with any previously generated stripesets. Utilizing these methods, G-SODP is able to generate 16 single overlap stripesets instead of the 20 generated by O-SODP. Note that, sometimes it's not always feasible to have any x accumulative distances that are different, given this we first ensure that one accumulative distance is different, then the two accumulative distances different and so on, which maximally reduces the multiple overlaps among stripesets.

Algorithm 2 presents the greedy algorithm pseudocode to generate base stripesets. Suppose a base stripeset contains $k + m$ disks, our goal is to select $k + m - 1$ disk distances. We start with the minimum distance 1 and inject it into an empty distance array. For any next distance (e.g., 2 or 3), we can put it before or after the existing distances in the array. The valid injection is to ensure no equal disk distance. For example, Figure 11 injects the next distance 3 at the beginning of the array, which leads to an equal distance of the sum of

next two distances (e.g., 3 = 2 + 1). In this case, the derived stripesets based on this base stripeset will cause two disks to be overlapping.

Algorithm 2: Greedy SODP (G-SODP)

Input: N, disks per server; s, stripe size
Output: B={$b_1, b_2, ...,$}, base stripesets
while *b = createBaseStripesets(B, N, s)* **do**
  $B \leftarrow b$
end
function CREATEBASESTRIPESETS(B, N, s)
b = [1]
for *i = 2: N-1* **do**
  for *j = 0 : length(b)* **do**
    b' = [b[0:j-1], i, b[j:end]]
    if *distances not in B* **then**
      if *distances in b' are not equal* **then**
        b = b'
        break
      end
    end
  end
  if *length(b) == s-1* **then**
    return b
  end
end
end function

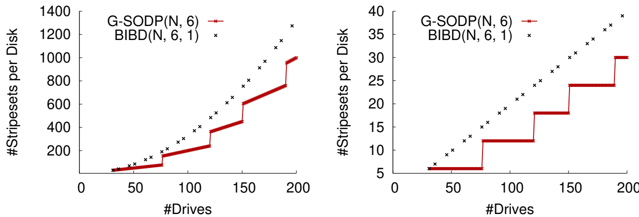


Fig. 12: Total number of stripesets and number of stripesets per disk for G-SODP and a defined BIBD configuration.

Figure 12 compares the total number of stripesets and stripesets per disk between BIBD and G-SODP. Here, BIBD(N,6,1) represents a full declustering layout over N disks, which only exists for a limited number of configurations. G-SODP aims to find a balanced layout for arbitrary disk size (e.g., $N > 31$). The comparison results show that G-SODP has fewer stripesets than BIBD(N,6,1), which indicates a gap between G-SODP and the full declustering BIBD. When a disk fails, G-SODP cannot guarantee that every surviving disk participates in that disks recovery (e.g., shorter rebuild time). However, it still attempts to maximize the rebuild performance in a balanced way, which in turn generates less stripesets than O-SODP to tolerate more concurrent disk failures.

COMPARISON OF O-SODP AND G-SODP Both O-SODP and G-SODP are able to generate single overlap stripesets in a balanced way. The only difference is O-SODP constructs perfectly balanced declustered layouts, where each pair-wise set of disks appears in exactly one stripeset. The perfect balance and exact overlap value of one make the number of stripesets minimized in the declustered layout. This design is not possible for all disk configurations (e.g., 8 total disks using 3-disk stripesets). G-SODP relaxes the overlap constraint slightly such that a few pair-wise combinations are not generated but a wider range of disk configurations are supported. Thus G-SODP provides greater configuration flexibility and fault tolerance while sacrificing a small amount of rebuild performance.

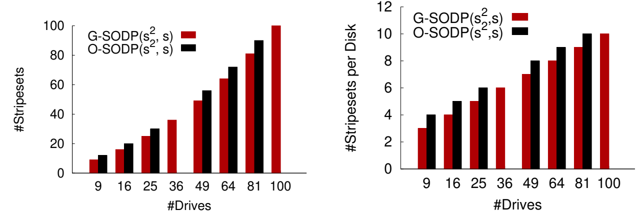


Fig. 13: Total number of stripesets and number of stripesets per disk for O-SODP and G-SODP.

Figure 13 compares the total number of stripesets and stripesets per disk between O-SODP and G-SODP. As previously stated, some configurations (e.g., 36 and 100 disks) are not supported in O-SODP, other configurations show similar results between G-SODP and O-SODP. When considering the number of stripesets per disk, G-SODP always generates one less stripeset than O-SODP does accounting for the small difference in total stripesets. In section V, we will provide a detailed comparison of O-SODP and G-SODP protecting against concurrent failures.

IV. SOL-SIM AND COFACTOR DESIGN

SOL-Sim is a discrete-event simulator that characterizes the reliability of erasure coded storage systems. Written in Python, SOL-Sim extends SimEDC [27] to supports additional erasure codes, chunk placement schemes, and data re-protection algorithms. Figure 14 shows the high-level SOL-Sim architecture which uses failure traces, disk layouts, and data protection schemes as input and returns timings and reliability metrics such as the probability of data loss (PDL) as output. SOL-Sim is designed to simulate reliability over longer periods of time (e.g., 5 years). One key component of the SOL-Sim architecture is the ability to use a complementary tool, CoFaCTOR, to evaluate multiple storage system designs over their entire lifetime easily.

A. SOL-Sim Architecture

In order to evaluate the data protection schemes in this paper we have combined the output of CoFaCTOR with simulation to evaluate the probability of data loss over a variety of realistic failure workloads. SOL-Sim stores all events in an event queue, which always returns the event with the smallest timestamp. If the event is a failure event, it will update the disk state, such as the clock and failure status. If the event

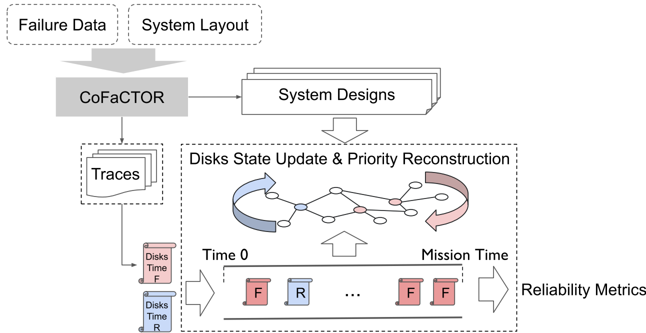


Fig. 14: SOL-Sim Architecture.

is a repair event it updates the corresponding disk's clock, repair status (e.g, critical to degraded or degraded to normal), and repair priority. SOL-Sim and CoFaCTOR provide the following features:

1) System Layout & System Failure Data: In order to generate a large set of realistic failure traces CoFaCTOR is seeded with both a system layout and a set of failure data collected from real system data. The system layout describes physical characteristics of the system that influence failure such as the physical position with the data center (data center row and rack number), the vertical position within the rack and even the disk positions within the storage enclosure (called the drawer row here to differentiate disks near the front of the enclosure, disks in the center of the enclosure, and disks near the rear of the enclosure). Detailed layout data in combination with positional data is critical in generating the set of failure traces and candidate system designs for SOL-Sim.

2) Failure Traces & System Designs: CoFaCTOR generates an arbitrarily large number of synthetic failure traces for use by SOL-Sim. These synthetic failure traces are generated using the survival analysis models seeded with real failure data. The second input into SOL-Sim is the set of system designs output by CoFaCTOR. These configurable system designs enable us to apply the generated failure traces to flexible system designs that explore both the physical system design space and the data protection algorithms used. For example, in this analysis we are able to alter the number of disks per enclosure (i.e. the failure domain for the declustered parity grouping) to explore future storage systems which are expected to be much denser than our existing system design.

3) Chunk Placement: SOL-Sim enables the use of multiple declustered placement algorithms and data protection schemes in conjunction with the failure data including: traditional RAID, complete declustered parity designs, dRAID, and both O-SODP and G-SODP depending on the availability of a single-overlap configuration for that design point.

4) Priority Reconstruction: SOL-Sim also implements a priority reconstruction algorithm that mimics those used in enterprise-grade production storage systems. If multiple drives fail within a server, to minimize data availability risk, any stripes that are missing two blocks are given priority for reconstruction. This approach is called critical reconstruction. After those critically affected stripes are reconstructed, the rest of the stripes continue to be reconstructed (called degraded reconstruction). While this algorithm is the state of the art it is not widely available for production systems.

B. CoFaCTOR Survival Analysis Model

While data on hard disk failures from Trinity can be used to test the reliability of data-redundancy patterns, generating synthetic failure streams with 'realistic' failures can asses the robustness of different approaches to unknown future failures. By 'realistic' in this case we mean that the failure times look similar in distribution to those observed previously on Trinity. We designed CoFaCTOR to generate these synthetic failure streams which are seeded by real failure data. A common assumption in reliability is to assume independent, identically distributed (iid) Weibull or Exponential failure times [8]; however, the exploratory analysis our Trinity failure traces showed correlation in failure rates with respect to the system configuration. In order to account for the relationship between the failure event times, we apply a two-parameter Weibull regression model, also commonly known as an accelerated failure time model [28]. Prior work has shown Weibull distributions to accurately reflect disk drive failures, both correlated and non-correlated [29]. The probability distribution function for the Weibull model is:

$$f(x) = -\ \frac{x}{-}^{\quad 1} e^{\ (x= \ )^{\rho}};$$

where is a scale parameter and is the shape parameter. The scale controls the size of the failure rate and the shape controls whether the failure rate increases, decreases, or stays constant over time. In the Weibull regression model, both the shape and scale are modeled as a linear function of covariates describing the system configuration:

$$\begin{aligned} &= \mathbf{X} \\ &= \mathbf{X} \quad ; \end{aligned} \tag{1}$$

where $\mathbf{X}$ is the matrix of covariate information, is the regression coefficient vector for , and is the same for . This model is fit using maximum likelihood using the *lifelines* package in Python [30].

We are able to relax the iid assumption because the Weibull parameters vary as a function of system configuration details described by the covariates. Given a set of regression coefficients [;], failure times for a drive with a particular covariate vector can be drawn as independent Weibull random variates. It is important to note, that in this model we are estimating component lifetimes. For Trinity, we focused on three covariates that indicated accelerate failure rates - file system ID, drawer row, and vertical position in the rack.

Figure 15 shows a comparison of simulated survival curves using the Weibull regression model (red) to the observed failure time data for two combinations of the covariate values. Specifically the blue line and shaded region show a nonparametric, Kaplan-Meier (KM) estimate [28] to the Trinity data and the 95% confidence interval for the survival curve. Qualitative difference between the KM estimate and the Weibull

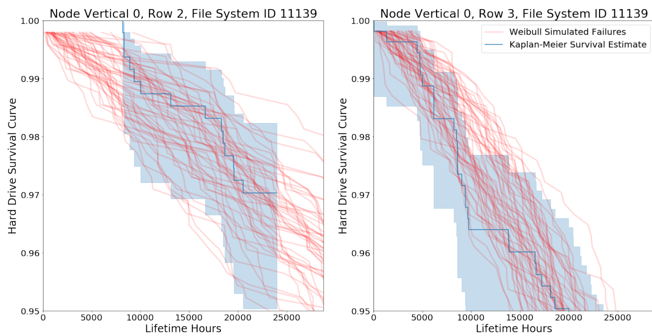


Fig. 15: Comparing 50 simulated survival curves from the fitted Weibull regression model to a nonparametric Kaplan-Meier estimate of the survival curve with collected data for two combinations of file system ID, node vertical position, and drawer row.

simulations illustrates areas in which the Weibull model is missing structure in the data. The simulated failure curves in Figure 15 look consistent with the Trinity data overall, with the regression model largely capturing the variation in lifetimes in drives with different covariate values. There is a sharp drop in the survival curve shortly before the 10,000 hour lifetime mark that is not consistent with the Weibull assumption. This drop was due to a batch of drives failing or being preemptively replaced due to a firmware issue. Addressing this is the scope of future work but its inclusion precludes the use of quantitative goodness-of-fit assessment.

V. Evaluation

A. Trinity Storage System Overview

LANL's Trinity file system is composed of two identical file systems accessible through the same set of gateway nodes within the Trinity platform. The identical file systems are organized as two parallel aisles of racks within our data center to both enable easier servicing/upgrades and protect against some types of failures external to the file systems. Each file system has 6 total metadata servers and 216 Lustre object storage servers (OSS) each with a single 41 disk Lustre object storage target (OST). OSS node pairs share a single two-drawer 84-bay disk chassis with 41 drives assigned to each of the OSS (the remaining 2 slots contain SSDs used as journal devices). Each drawer within the 84 disk enclosure is composed of 3 rows with 14 drive slots per row. Row 1 holds the 14 drives nearest the front of the drawer and row 3 holds the 13 drives and the SSD at the rear of the drawer. The 41 disk OSTs use a declustered parity approach to construct 8+2 protected stripes with 128KiB stripes forming a 1MiB stripe set. The simulations presented in this paper assume that all drives are 6TB Seagate Makarra drives, the file system is at 60% capacity utilization, and the rebuild disk bandwidth has been set at approximately 50MB/s per disk [31].

B. Simulation Failure Traces

To create a sufficient number of failure streams to perform our reliability analysis we use the trinity file system configuration, two years of drive failure data, and CoFaCTOR to generate 10000 5-year long failure traces. We note in individual experiments where the system design has been altered to explore alternative storage system designs (e.g. changes in disk capacity, disk enclosure size, or total number of drives). We also use CoFaCTOR to generate 10000 traces where a fixed percentage of disks fail instantaneously or over a fixed period of time (using Poisson arrivals) to simulate catastrophic failure events such as power outages.

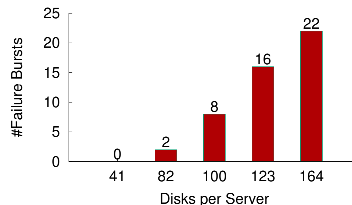


Fig. 16: The number of 3 disk failures occurring within an 8 hour window using traces generated by CoFaCTOR.

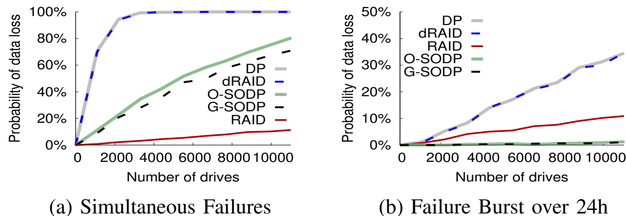
C. System Lifetime Failure Analysis

In order to study the correlated failures, we investigate the number of failure bursts (e.g., multiple failures in a failure domain within 8 hours) with varied numbers of disks per server. As shown in Figure 16, among 10000 failure traces, larger disk enclosure sizes are more likely to encounter failure bursts and possibly data loss.

Disks per Server	Scheme	Spares	6TB	14TB	20TB
123	DP	1	0	4	7
		2	0	4	7
		3	0	4	7
	SODP	1	0	0	0
164	DP	1	1	2	5
		2	0	1	4
		3	0	1	4
	SODP	1	1	1	1

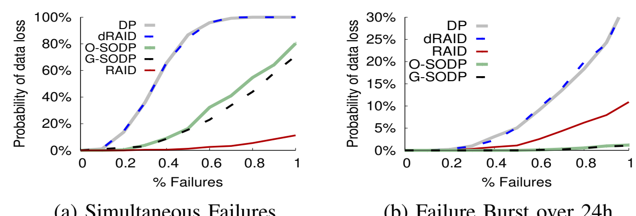
TABLE II: Comparison of the number of 5-year traces with at least one data loss event. We evaluate different disks per server, data protection schemes, and the number of distributed spares for differing disk capacities.

Figure 16 shows the likelihood of a failure burst as we alter the number of disks allocated to each server. Not surprisingly, as we increase the number of disks per server the number of failure bursts over the life of a storage system increases. Table II then shows how well Trinity's declustered parity scheme protects against data loss events compared with an SODP data placement scheme while varying the spare capacity and drive capacity. We use the 123 and 164 disk configurations with G-SODP because an insufficient number of disks exist for an RS(8,2) SODP configuration at the smaller disk counts. We see that SODP prevents data loss with a lower sparing overhead better than the existing Trinity file system, however a rapid burst of failures within a single stripe before copyback completes still causes a single data loss event in the 164 disk configuration.



(a) Simultaneous Failures (b) Failure Burst over 24h

Fig. 17: The probability of data loss with failure of 1% of the drive population as the number of disk drives are scaled. In 17a we see that if the drive loss is instantaneous a non-overlapping RAID scheme provides the greatest fault tolerance. However in 17b we distribute the failures over a 24 hour period which allows the fast rebuild performance of declustered schemes to greatly reduces the overall probability of data loss.



(a) Simultaneous Failures (b) Failure Burst over 24h

Fig. 18: The probability of data loss varied with the percentage of failed disks in a population of 11000 drives. During instantaneous failures in 18a traditional declustered parity schemes experience a 100% chance of data loss once approximately 0.6% of the drives have failed. When the failures are distributed over 24 hours SODP schemes exhibit a less than 1.2% chance of data loss.

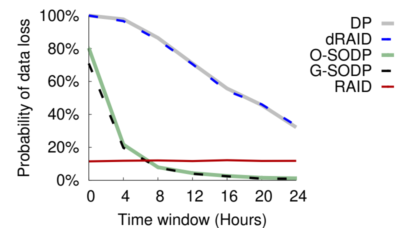


Fig. 19: The probability of data loss as 1% of drive failures are distributed over a longer time scale. As the failure window extends beyond the time to rebuild a disk we see that the SODP schemes provide better probability of data loss than non-overlapping RAID schemes. G-SODP provides slightly lower probabilities of data loss compared to O-SODP.

D. Catastrophic Failures Analysis

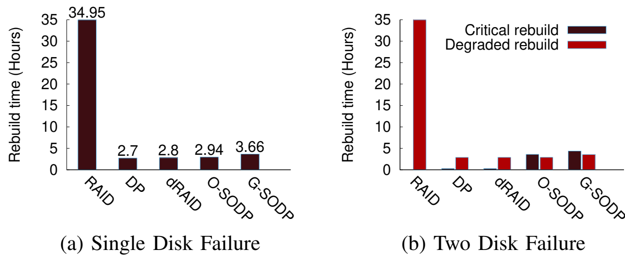
To explore how different data protection schemes perform during common burst failure scenarios (e.g. a power outage or cascading failure that results in a large number of drives failing in a short time window) we simulated failures correlated in time but not correlated in any other dimension. We also compare the corresponding rebuild performance for schemes including traditional RAID, dRAID, Trinity's declustered parity (DP), O-SODP and G-SODP.

1) Effect of Disk Population Size: Figure 17 shows the probability of data loss (PDL) during a burst failure of 1% disk failures and increased number of disks. In particular, we increase the total number of disks from 1100 to 11,000 and examine the PDL for 1% drive population failure instantaneously and randomly distributed over 24 hours. In Figure 17(a), failures occur in an instantaneous burst and rebuild time is irrelevant. Non-overlapping RAID can tolerate the most simultaneous failures with 11.3% PDL. The greater fault tolerance of the SODP schemes protects data at smaller disk counts but is not sufficient for large disk populaitons. With Trinity's declustered parity and dRAID we see a 100% chance of data loss over 6600 disks. In Figure 17(b), the failures simulate a cascading failure occurring over 24 hours, thus data is rebuilt during the 24-hour failure period for the declustered parity schemes. By tolerating more failures and having fast rebuild performance, G-SODP and O-SODP have 1.05% and 1.2% PDL for 11,000 disks, respectively. Trinity's declustered parity and dRAID also benefit from fast rebuild performance, but the lack of greater fault tolerance lowers the PDL to only 34.6%. Interestingly, even with failures distributed over 24 hours RAID is better than Trinity's declustered parity and dRAID indicating that in this experiment fault tolerance is more important than rebuild performance.

2) Effect of Burst Size: Figure 18 varies the number of failed disks in a population of 11,000 drives from 0% to 1% to compare each of the parity placement schemes. As before we examine the probability of data loss (PDL) for simultaneous failures and failures over 24 hours. From Figure 18(a) it's obvious that the probability of data loss increases with the percentage of simultaneous failures. At approximately 0:6% of the total drive population failed the PDL of Trinity declustered parity and dRAID reach 100%, while O-SODP experiences 32.3% PDL and G-SODP has 23% PDL. RAID again tolerates the most simultaneous failures with a 2:6% chance of data loss. Figure 18(b) shows the probability of data loss with the failures distributed over 24 hours. We see that the SODP have the lowest chance of data loss due to high fault tolerance and fast rebuilds. Even in the case of 1% failures, O-SODP and G-SODP have 1.2% and 1.05% PDL, respectively. Because no rebuilds completed in 24 hours RAID experienced the same PDL as Figure 18(a) while both Trinity declustered parity and dRAID shows a great reduction in PDL due to their faster rebuild performance.

Finally, we investigate how the time window over which the failures occur effects the probability of data loss. Figure 19 shows the probability of data loss in 11,000 drives in the presence of 1% failures distributed over varying timespans (using a Poission arrival process for failures). Because 24 hours is greater than the rebuild time for RAID the PDL remains unchanged within the varying time period. However, as declustered rebuilds complete in 2-3 hours we see the PDLs of Trinity declustered parity and dRAID are reduced rapidly. Even with slower lower rebuild performance for O-SODP, G-SODP we see extremely low probabilities of data loss.

3) Comparison of Rebuild Performance: Figure 20(a) compares the disk rebuild time by injecting a random single

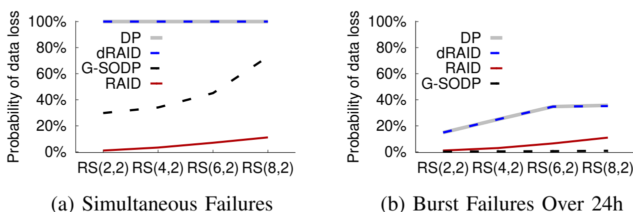


(a) Single Disk Failure (b) Two Disk Failure

Fig. 20: Rebuild times for single and two disk failures

disk failure. Trinity declustered parity and dRAID show the fastest rebuild times requiring only 2.7 and 2.8 hours. O-SODP exhibits similar rebuild performance to DP and dRAID while G-SODP requires slightly more rebuild time. Because it lacks declustering traditional raid requires greater than 24 hours to perform a rebuild. With multiple disk failures we exploit the priority reconstruction algorithm which give vulnerable stripes higher rebuild priority to avoid data loss. In our configuration (e.g., 8 + 2), the maximal tolerable failures within a single stripeset is 2, therefore, Figure 20(b) compares the critical rebuild (e.g., stripes with 2 failures) and degraded rebuild (e.g., stripes with 1 failures) time during reconstruction process. We observe that DP, dRAID and O-SODP have almost the same degraded rebuild time due to full declustering. A drawback of the SODP schemes is the longer critical rebuild times. Finally, RAID systems do not use priority reconstruction.

4) Comparison of Storage Efficiency: Finally, we study the effects of storage efficiency by using different parity configurations. Since O-SODP cannot be calculated for arbitrary configurations we only compare RAID, DP, dRAID and G-SODP. In Figure 21 we show how the parity overhead effects the probability of data loss for differing parity overheads.Generally we see that additional parity overhead is moderately effective during failure bursts, but that during failures distributed over time even extremely low overhead G-SODP configurations provide low probabilities of data loss.



(a) Simultaneous Failures (b) Burst Failures Over 24h

Fig. 21: The probability of data loss for RAID, DP, dRAID, G-SODP under simultaneous failures with varying parity overheads. While (a) shows that additional parity overhead is moderately effective at improving data loss probabilities during burst failures, (b) shows that G-SODP provides low probabilities of data loss while using low parity overheads.

VI. Conclusions

In our re-examination of declustered parity data placement schemes it is apparent that existing declustered parity data protection schemes are not designed to tolerate the correlated failure bursts becoming increasingly common in cloud and HPC data centers. To that end, we have proposed adding 2 new criteria to the design principles for declustered parity designs:

> Maximizing the number of simultaneous disk failures tolerated without increasing parity overhead, and
> Minimizing disk rebuild time by balancing parity stripes across all disks.

To balance both of these criteria equally we proposed Single-Overlap Declustered Parity, a data placement scheme that minimizes rebuild time and tolerates more failed disks than existing declustered parity designs. However, in order to build a flexible algorithm to generate SODP stripesets it became necessary to relax the single-overlap requirement and allow a small number of stripesets that do not overlap all other stripesets. Surprisingly, this more flexible algorithm demonstrated the lowest data loss during failures occurring within small windows of time. In our experiments we showed that adding disks to a declustered placement group increases the probability of a data loss event by a greater than linear amount when faced with correlated failures. But the additional disk results in a less than linear improvement in rebuild time because the amount of data being rebuilt remains fixed even as the rebuild rate is proportionally increased. Thus a data protection scheme, such as Greedy Single-Overlap Declustered Parity, which is designed to trade small amounts of rebuild performance in order to tolerate a large number of disk failures can reduce the probability of data loss substantially (30x in some of our test configurations).

In future work we plan a detailed evaluation of the read and write performance of SODP-based data protection schemes and a further exploration of how to optimize the tradeoffs between rebuild performance and disk failure tolerance. We also plan to apply our design principles for improving data protection schemes to distributed storage systems that leverage multiple levels of erasure coding and replication.

VII. Acknowledgements

The authors thank John Bent for helpful discussions on priority rebuild algorithms within declustered parity storage systems. We also thank Data Direct Networks for their support and co-funding this work as partners in the Efficient Mission Centric Computing Consortium (EMC3). The university authors were supported by funding from NSF(grant #CNS-1563956). This manuscript has been approved for unlimited release and has been assigned LA-UR-20-23191. This work has been co-authored by an employee of Triad National Security, LLC which operates Los Alamos National Laboratory under Contract No. 89233218CNA000001 with the U.S. Department of Energy/National Nuclear Security Administration. The publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of the manuscript, or allow others to do so, for United States Government purposes.

REFERENCES

[1] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. Finding a needle in haystack: Facebook's photo storage. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 47–60, Berkeley, CA, USA, 2010. USENIX Association.

[2] Subramanian Muralidhar, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Viswanath Sivakumar, Linpeng Tang, and Sanjeev Kumar. f4: Facebook's warm BLOB storage system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 383–398, Broomfield, CO, October 2014. USENIX Association.

[3] S. Oral, J. Simmons, J. Hill, D. Leverman, F. Wang, M. Ezell, R. Miller, D. Fuller, R. Gunasekaran, Y. Kim, S. Gupta, D. T. S. S. Vazhkudai, J. H. Rogers, D. Dillow, G. M. Shipman, and A. S. Bland. Best practices and lessons learned from deploying and operating large-scale data-centric parallel file systems. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 217–228, Nov 2014.

[4] Glenn K. Lockwood, Kirill Lozinskiy, Lisa Gerhardt, Ravi Cheema, Damian Hazen, and Nicholas J. Wright. Designing an all-flash Lustre file system for the 2020 NERSC Perlmutter system. In *Proceedings of the 2019 Cray User Group (CUG)*, 2019.

[5] Ao Ma, Rachel Traylor, Fred Douglis, Mark Chamness, Guanlin Lu, Darren Sawyer, Surendar Chandra, and Windsor Hsu. Raidshield: characterizing, monitoring, and proactively protecting against disk failures. *ACM Transactions on Storage (TOS)*, 11(4):17, 2015.

[6] Seagate Technology LLC. Seagate cloud storage array, January 2018.

[7] Seagate Technology LLC. Exos e 4U106, January 2018.

[8] Kevin M. Greenan, James S. Plank, and Jay J. Wylie. Mean time to meaningless: Mttdl, markov models, and storage system reliability. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'10, pages 5–5, Berkeley, CA, USA, 2010. USENIX Association.

[9] Saurabh Kadekodi, K. V. Rashmi, and Gregory R. Ganger. Cluster storage systems gotta have heart: improving storage efficiency by exploiting disk-reliability heterogeneity. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 345–358, Boston, MA, February 2019. USENIX Association.

[10] Asaf Cidon, Stephen Rumble, Ryan Stutsman, Sachin Katti, John Ousterhout, and Mendel Rosenblum. Copysets: Reducing the frequency of data loss in cloud storage. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 37–48, 2013.

[11] Ramnatthan Alagappan, Aishwarya Ganesan, Yuvraj Patel, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Correlated crash vulnerabilities. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 151–167, Berkeley, CA, USA, 2016. USENIX Association.

[12] Daniel Ford, François Labelle, Florentina I. Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. Availability in globally distributed storage systems. In *Presented as part of the 9th USENIX Symposium on Operating Systems Design and Implementation*, Vancouver, BC, 2010. USENIX.

[13] James Lujan, Manuel Vigil, Garrett Kenyon, Karissa Sanbonmatsu, and Brian Albright. Trinity supercomputer now fully operational.

[14] Mark Holland and Garth A. Gibson. Parity declustering for continuous operation in redundant disk arrays. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS V, pages 23–35, New York, NY, USA, 1992. ACM.

[15] Guillermo A Alvarez, Walter A Burkhard, and Flaviu Cristian. Tolerating multiple failures in raid architectures with optimal storage and uniform declustering. In *ACM SIGARCH Computer Architecture News*, volume 25, pages 62–72. ACM, 1997.

[16] John Fragalla. Improving Lustre OST performance with ClusterStor GridRAID. In *2014 HPCAC Stanford HPC Exascale Conference*, 2014.

[17] GA Alverez, Walter A Burkhard, LL Stockmeyer, and Flaviu Cristian. Declustered disk array architectures with optimal and near-optimal parallelism. In *Proceedings. 25th Annual International Symposium on Computer Architecture (Cat. No. 98CB36235)*, pages 109–120. IEEE, 1998.

[18] Thomas JE Schwarz, Jesse Steinberg, and Walter A Burkhard. Permutation development data layout (pddl). In *Proceedings Fifth International Symposium on High-Performance Computer Architecture*, pages 214–217. IEEE, 1999.

[19] Zfs draid. https://github.com/openzfs/zfs/wiki/dRAID-HOWTO, 2016.

[20] Guangyan Zhang, Zican Huang, Xiaosong Ma, Songlin Yang, Zhufan Wang, and Weimin Zheng. Raid+: deterministic and balanced data distribution for large disk enclosures. In *16th fUSENIXg Conference on File and Storage Technologies (fFASTg 18)*, pages 279–294, 2018.

[21] Neng Wang, Yinlong Xu, Yongkun Li, and Si Wu. Oi-raid: a two-layer raid architecture towards fast recovery and high reliability. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 61–72. IEEE, 2016.

[22] Zhipeng Li, Min Lv, Yinlong Xu, Yongkun Li, and Liangliang Xu. D3: Deterministic data distribution for efficient data reconstruction in erasure-coded distributed storage systems. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 545–556. IEEE, 2019.

[23] Richard R Muntz and John CS Lui. *Performance analysis of disk arrays under failure*. Computer Science Department, University of California, 1990.

[24] Jeff Bonwick, Matt Ahrens, Val Henson, Mark Maybee, and Mark Shellenbaum. The zettabyte file system. Technical report, Sun Microsystems, 2003.

[25] Eric J Schwabe and Ian M Sutherland. Improved parity-declustered layouts for disk arrays. *journal of computer and system sciences*, 53(3):328–343, 1996.

[26] Huan Ke. Optimal Single Overlap Declustered Parity (O-SODP) Feasible Solutions. shorturl.at/hqMR0.

[27] Mi Zhang, Shujie Han, and Patrick PC Lee. A simulation analysis of reliability in erasure-coded data centers. In *2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS)*, pages 144–153. IEEE, 2017.

[28] John P Klein and Melvin L Moeschberger. *Survival analysis: techniques for censored and truncated data*. Springer Science & Business Media, 2006.

[29] Bianca Schroeder and Garth A. Gibson. Disk failures in the real world: What does an mttf of 1,000,000 hours mean to you? In *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, FAST '07, Berkeley, CA, USA, 2007. USENIX Association.

[30] Cameron Davidson-Pilon, Jonas Kalderstam, Paul Zivich, Ben Kuhn, Andrew Fiore-Gartland, Luis Moneda, Gabriel, Daniel WIlson, Alex Parij, Kyle Stark, Steven Anton, Lilian Besson, Jona, Harsh Gadgil, Dave Golland, Sean Hussey, Ravin Kumar, Javad Noorbakhsh, Andreas Klintberg, Eduardo Ochoa, Dylan Albrecht, dhuynh, Dmitry Medvinsky, Denis Zgonjanin, Daniel S. Katz, Daniel Chen, Christopher Ahern, Chris Fournier, Arturo, and André F. Rendeiro. Camdavidsonpilon/lifelines: v0.22.3 (late), August 2019.

[31] Mark Swan. Sonexion GridRAID characteristics. In *Proceedings of the 2014 Cray User Group (CUG)*, 2014.