Parity Lost and Parity Regained

Andrew Krioukov*, Lakshmi N. Bairavasundaram*, Garth R. Goodson†, Kiran Srinivasan†,
Randy Thelen†, Andrea C. Arpaci-Dusseau*, Remzi H. Arpaci-Dusseau*
**University of Wisconsin-Madison* †*Network Appliance, Inc.*
{*krioukov, laksh, dusseau, remzi*}*@cs.wisc.edu,* {*goodson, skiran, rthelen*} *@ netapp.com*

Abstract

RAID storage systems protect data from storage errors, such as data corruption, using a set of one or more integrity techniques, such as checksums. The exact protection offered by certain techniques or a combination of techniques is sometimes unclear. We introduce and apply a formal method of analyzing the design of data protection strategies. Specifically, we use model checking to evaluate whether common protection techniques used in parity-based RAID systems are sufficient in light of the increasingly complex failure modes of modern disk drives. We evaluate the approaches taken by a number of real systems under single-error conditions, and find flaws in every scheme. In particular, we identify a parity pollution problem that spreads corrupt data (the result of a single error) across multiple disks, thus leading to data loss or corruption. We further identify which protection measures must be used to avoid such problems. Finally, we show how to combine real-world failure data with the results from the model checker to estimate the actual likelihood of data loss of different protection strategies.

1 Introduction

Data reliability and integrity is vital to storage systems. Performance problems can be tuned, tools can be added to cope with management issues, but data loss is seen as catastrophic. As Keeton *et al.* state, data unavailability may cost a company "... more than $1 million/hour", but the price of data loss is "even higher" [23].

In well-designed, high-end systems, disk-related errors are still one of the main causes of potential trouble and thus must be carefully considered to avoid data loss [25]. Fortunately, with simple disk errors (*e.g.*, an entire disk failing in a fail-stop fashion), designing protection schemes to cope with disk errors is not overly challenging. For example, early systems successfully handle the failure of a single disk through the use of mirroring or parity-based redundancy schemes [6, 24, 29]. Although getting an implementation to work correctly may be challenging (often involving hundreds of thousands of lines of code [38]), one could feel confident that the design properly handles the expected errors.

Unfortunately, storage systems today are confronted with a much richer landscape of storage errors, thus considerably complicating the construction of correctly-designed protection strategies. For example, disks (and other storage subsystem components) are known to exhibit latent sector errors, corruption, lost writes, misdirected writes, and a number of other subtle problems during otherwise normal operation [2, 3, 17, 21, 30, 37]. Thus, a fully-formed protection strategy must consider these errors and protect data despite their occurrence.

A number of techniques have been developed over time to cope with errors such as these. For example, various forms of checksumming can be used to detect corruption [4, 35]; combined with redundancy (*e.g.*, mirrors or parity), checksumming enables both the detection of and recovery from certain classes of errors. However, given the broad range of techniques (including sector checksums, block checksums, parental checksums, write-verify operations, identity information, and disk scrubbing, to list a few), exactly which strategies protect against which errors is sometimes unclear; worse, combining different approaches in a single system may lead to unexpected gaps in data protection.

We propose a more formal approach based on model checking [20] to analyze the design of protection schemes in modern storage systems. We develop and apply a simple *model checker* to examine different data protection schemes. Within the system, one first implements a simple logical version of the protection strategy under test; the model checker then applies different sequences of read, write, and error events to exhaustively explore the state space of the system, either producing a chain of events that lead to data loss or a "proof" that the scheme works as desired.

We use the model checker to evaluate a number of different approaches found in real RAID systems, focusing on parity-based protection and single errors. We find holes in all of the schemes examined, where systems potentially exposes data to loss or returns corrupt data to the user. In data loss scenarios, the error is detected, but the data cannot be recovered, while in the rest, the error is not detected and therefore corrupt data is returned to the user. For example, we examine a combination of two techniques – block-level checksums (where checksums of the data block are stored within the same disk block as data and verified on every read) and write-verify (where data is read back immediately after it is written to disk and verified for correctness), and show that the scheme could still fail to detect certain error conditions, thus returning corrupt data to the user.

We discover one particularly interesting and general problem that we call *parity pollution*. In this situation, corrupt data in one block of a stripe spreads to other blocks through various parity calculations. We find a number of cases where parity pollution occurs, and show how pollution can lead to data loss. Specifically, we find that data scrubbing (which is used to reduce the chances of double disk failures) tends to be one of the main causes of parity pollution.

We construct a protection scheme to address all issues we discover including parity pollution. The scheme uses a version-mirroring technique in combination with block-level checksums and physical and logical identity information, leading to a system that is robust to a full and realistic range of storage errors.

With analyses of each scheme in hand, we also show how a system designer can combine real data of error probability with our model checker's results to arrive upon a final estimation of data loss probability. Doing so enables one to compare different protection approaches and determine which is best given the current environment. An interesting observation that emerges from the probability estimations is the trade-off between a higher probability detected data loss versus a lower probability of undetectable data corruption. For example, this trade-off is relevant when one decides between storing checksums in the data block itself versus storing them in a parent block. Another interesting observation is that data scrubbing actually increases the probability of data loss significantly under a single disk error.

The rest of the paper is structured as follows. Section 2 discusses background, while Section 3 describes our approach to model checking. Section 4 presents the results of using the model checker to deconstruct a variety of protection schemes; Section 5 presents the results of our probability analysis of each scheme combined with real-world failure data. Section 6 describes related work and Section 7 concludes.

2 Background

We provide some background first on a number of protection techniques found in real systems, and then on the types of storage errors one might expect to see in modern systems.

2.1 Protection Techniques

Protection techniques have evolved greatly over time. Early multiple disk systems focused almost solely on recovery from entire disk failures; detection was performed by the controller, and redundancy (*e.g.*, mirrors or parity) was used to reconstruct data on the failed disk [12].

Unfortunately, as disk drives became bigger, faster, and cheaper, new and interesting failure modes began to appear. For example, Network Appliance$^{\mathrm{TM}}$ recently added protection against "lost writes" [37], *i.e.*, write requests that appear to have been completed by the disk, but (for some reason) do not appear on the media. Many other systems do not (yet) have such protections, and the importance of such protection is difficult to gauge.

This anecdote serves to illustrate the organic nature of data protection. While it would be optimal to simply write down a set of assumptions about the fault model and then design a system to handle the expected errors, in practice such an approach is not practical. Disks (and other storage subsystem components) provide an ever-moving target; tomorrow's disk errors may not be present today. Worse, as new problems arise, they must be incorporated into existing schemes, rather than attacked from first principles. This aspect of data protection motivates the need for a formal and rigorous approach to help understand the exact protection offered by combinations of techniques.

Table 1 shows the protection schemes employed by a range of modern systems. Although the table may be incomplete (*e.g.*, a given system may use more than the protections we list, as we only list what is readily made public via published papers, web sites, and documentation), it hints at the breadth of approaches employed as well as the on-going development of protection techniques. We discuss each of these techniques in more detail in Section 4, where we use the model checker to determine their efficacy in guarding against storage errors.

2.2 Storage Errors

We now discuss the different types of storage errors. Many of these have been discussed in detail elsewhere [2, 3, 30, 37]. Here, we provide a brief overview and discuss their frequency of occurrence (if known).

- **Latent sector errors**: These errors occur when data cannot be reliably read from the disk drive medium.

System	RAID	Scrubbing	Sector checksums	Block checksums	Parent checksums	Write Verify	Physical Identity	Logical Identity	Version Mirroring	Other
Hardware RAID card (say, Adaptec™ 2200 S [1])	√									
Linux software RAID [16, 28]	√	√								
Pilot [31]								√		√
Tandem NonStop® [4]	√		√				√			
Dell™ Powervault™ [14]	√	√	√							√
Hitachi Thunder 9500™ [18, 19]	√		√			√				
NetApp® Data ONTAP® [37]	√	√		√		√	√	√		
ZFS [36] with RAID-4	√	√			√					

Table 1: **Protections in Real Systems.** *This table shows the known protections used in real-world systems. Some systems have other protections: Pilot uses a scavenger routine to recover metadata, and Powervault uses a 1-bit "write stamp" and a timestamp value to detect data-parity mismatches. Systems may use further protections (details not made public).*

The disk drive returns an explicit error code to the system when a latent sector error is encountered.

- **Corruptions**: As the name indicates, these errors are said to occur when the data stored in a disk block is corrupted by an element of the storage stack.

- **Torn writes**: Disk drives may end up writing only a portion of the sectors in a given write request. Often, this occurs when the drive is power-cycled in the middle of processing the write request.

- **Lost writes**: In rare cases, buggy firmware components may return a success code to indicate completion of a write, but not perform the write in reality.

- **Misdirected writes**: In other rare cases, buggy firmware may write the data to the wrong disk or the wrong location within a disk. The effect of this error is two-fold: the original disk location does not receive the write it is supposed to receive (lost write), while the data in a different location is overwritten (with effects similar to corruption or lost write).

Latent sector errors affect about 19% of nearline and about 2% of enterprise class disks within 2 years of use [2]. Corruptions or torn writes affect on average around 0.6% of nearline and 0.06% of enterprise class disks within 17 months of use [3]. Lost or misdirected writes occur in about 0.04% of nearline and 0.007% of enterprise class disks within the first 17 months of use [3]. While the lost write numbers seem rather low, it is important to note that when a company sells a few million disks, at least one (and likely many more) customers could be affected by lost writes every year.

2.3 Error Outcomes

Depending on the protection techniques in place, storage systems errors may have one or more of the following outcomes:

- **Data recovery:** The scenario where the protection strategy detects the error, and uses parity to successfully recover data.

- **Data loss:** The scenario where the protection strategy detects the error, but is unable to successfully recover data. In this case, the storage system reports an error to the user.

- **Corrupt data:** The scenario where the protection strategy does not detect the error, and therefore returns corrupt data to the user.

3 Model Checking

We have developed a simple model checker to analyze the design of various data protection schemes. The goal of the model checker is to identify all execution sequences, consisting of user-level operations, protection operations, and disk errors, that can lead to either data loss or corrupt data being returned to the user. The model checker exhaustively evaluates all possible states of a single RAID stripe by taking into account the effects of all possible operations and disk errors for each state.

We have chosen to build our own model checker instead of using an existing one since it is easier to build a simple model checker that is highly specific to RAID data protection; for example, the model checker assumes that the data disks are inter-changeable, thereby reducing the number of unique states. However, there is no fundamental reason why our analysis cannot be performed on a different model checker.

Models for the model checker are built on top of some basic primitives. A RAID stripe consists of N disk blocks where the contents of each disk block is defined by the model using primitive components consisting of user data entries and protections. Since both the choice of components and their on-disk layout affect the data reliability, the model must specify each block as a series of entries (corresponding to sectors within a block). Each entry can be atomically read or written.

The model checker assumes that the desired unit of consistency is one disk block. All protection schemes are evaluated with this assumption as a basis.

3.1 Model Checker Primitives

The model checker provides the following primitives for use by the protection scheme.

- Disk operations: The conventional operations disk read and disk write are provided. These operations are atomic for each entry (sector) and not over multiple entries that form a disk block.

- Data protection: The model checker and the model in conjunction implement various protection techniques. The model checker uses model-specified knowledge of the protections to evaluate different states. For example, the result of checksum verification is part of the system state that is maintained by the model checker. Protections like parity and checksums are modeled in such a way that "collisions" do not occur; we wish evaluate the spirit of the protection, not the choice of hash function.

The model defines operations such as user read and user write based on the model checker primitives. For instance, a user write that writes a part of the RAID stripe will be implemented by the model using disk read and disk write operations, parity calculation primitives, and protection checks.

3.2 Modeling Errors

The model checker injects exactly one error during the analysis of the protection scheme. The different types of storage errors discussed in Section 2.2 are supported. We now describe how the different errors are modeled.

- Latent sector errors: These errors are modeled as inaccessible data – an explicit error is returned when an attempt is made to read the disk block. Disk writes always succeed; it is assumed that if a latent sector error occurs, the disk automatically *remaps* [2] the sectors.

- Corruptions: These errors are modeled as a change in value of a disk sector that produces a new value (*i.e.*, no collisions).

- Lost writes: These errors are modeled by not updating any of the sectors that form a disk block when a subsequent disk write is issued.

- Torn writes: These errors are modeled by updating only a portion of the sectors that form a disk block when a subsequent disk write is issued.

- Misdirected writes: These errors manifest in two ways: (i) they appear as a lost write for the block the write was intended to (the target), and (ii) it overwrites a different disk location (the victim). We assume that the target and victim are on different RAID stripes (otherwise, it would be a double error), and therefore can be modeled separately. Thus, we need to model only the victim, since the effects of a lost write on the target is an error we already study. A further assumption we make is that the data being written is block-aligned with the victim. Thus, a misdirected write is modeled by performing a write to a disk block (with valid entries) without an actual request from the model.

3.3 Model Checker States

A state according to the model checker is defined using the following sub-states: (a) the validity of each data item stored in the data disks as maintained by the model checker, (b) the results of performing each of the protection checks of the model, and (c) whether valid data and metadata items can be regenerated from parity for each of the data disks. The data disks are considered interchangeable; for example, data disk $D0$ with corrupt data is the same as data disk $D1$ with corrupt data as long as all other data and parity items are valid in both cases. As with any model checker, the previously explored states are remembered to avoid re-exploration.

The output of the model checker is a state machine that starts with the RAID stripe in the clean state and contains state transitions to each of the unique states discovered by the model checker. Table 2 contains a list of operations and errors that cause the state transitions.

4 Analysis

We now analyze various protection schemes using the model checker. We add protection techniques – RAID,

Operation	Description	Notation
User read	Read for any data disk	$\mathrm{R(X)}$
User write	Write for any combination of disks in the stripe (the model performs any disk reads needed for parity calculation)	$\mathrm{W()}$ is any write, $\mathrm{W_{ADD}()}$ is write with additive parity, $\mathrm{W_{SUB}()}$ is subractive; Parameters: X+ is "data disk X plus others", !X is "other than data disk X", full is "full stripe"
Scrub	Read all disks, verify protections, recompute parity from data, and compare with on-disk parity	S
Latent sector error	Disk read to a disk returns failure	$\mathrm{F_{LSE}(X)}$, $\mathrm{F_{LSE}(P)}$ for data disk X and parity disk respectively
Corruption	A new value is assigned to a sector	$\mathrm{F_{CORRUPT}(X)}$, $\mathrm{F_{CORRUPT}(P)}$ for data disk X and parity disk respectively
Lost write	Disk write issued is not performed, but success is reported	$\mathrm{F_{LOST}(X)}$, $\mathrm{F_{LOST}(P)}$ for data disk X and parity disk respectively
Torn write	Only the first sector of a disk write is written, but success is reported	$\mathrm{F_{TORN}(X)}$, $\mathrm{F_{TORN}(P)}$ for data disk X and parity disk respectively
Misdirected write	A disk block is overwritten with data following the same layout as the block, but not meant for it	$\mathrm{F_{MISDIR}(X)}$, $\mathrm{F_{MISDIR}(P)}$ for data disk X and parity disk respectively

Table 2: **Model Operations.** *This table shows the different sources of state transitions: (a) operations that are performed on the model, and (b) the different errors that are injected.*

data scrubbing, checksums, write-verify, identity, version mirroring – one by one, and evaluate each setup. We restrict our analysis to the protection offered by the different schemes against a single error. Indeed, we find that most schemes cannot recover from even a single error (given the proper failure scenario).

4.1 Bare-bones RAID

The simplest of protection schemes is the use of parity to recover from errors. This type of scheme is traditionally available through RAID hardware cards [1]. In this scheme, errors are typically detected based on error codes returned by the disk drive.

Figure 1 presents the model of bare-bones RAID, specified using the primitives provided by the model checker. In this model, a user read command simply calls a RAID-level read, which in turn issues a disk read for all disks. The disk read primitive returns the "data" successfully unless a latent sector error is encountered. On a latent sector error, the RAID read routine calls the reconstruct routine, which reads the rest of the disks, and recovers data through parity calculation. At the end of a user read, in place of returning data to the user, a validity check primitive is called. This model checker primitive verifies that the data is indeed valid; if it is not valid, then the model checker has found a hole in the protection scheme that returns corrupt data to the user.

When one or more data disks are written, parity is recalculated. Unless the entire stripe is written, parity calculation requires disk reads. In order to optimize the number of disk reads, parity calculation may be performed in an additive or subtractive manner. In additive parity calculation, data disks other than the disks being written are read and the new parity is calculated as the XOR over the read blocks and the blocks being written. In subtractive parity calculation, the old data in the disks being written and the old parity are first read. Then, the new parity is the XOR of old data, old parity, and new data. Since parity calculation uses data on disk, it should verify the data read from disk. We shall see in the subsections that follow that the absence of this verification could violate data protection.

When the model checker is used to evaluate this model and only one disk error is injected, we obtain the state machine shown in Figure 2. Note that the state machine shows only those operations that result in state transitions (*i.e.*, self-loops are omitted). The model starts in the `clean` state and transitions to different states when errors occur. For example, a latent sector error to data disk X places the model in state `Disk`$_X$ `LSE`. The model transitions back to `clean` state, when one of the following occurs: (a) user read to data disk X i.e. $\mathrm{R(X)}$, (b) user write to data disk X plus 0 or more other disks that in turn causes a disk read to data disk X for subtractive parity calculation ($\mathrm{W_{SUB}(X+)}$), and (c) user write to any disks that result in additive parity calculation, thereby either causing data disk X to be read or data disk X to be overwritten ($\mathrm{W_{ADD}()}$). Thus, we see that the model can recover from a latent sector error to data disks. We also see that the model can recover from a latent sector error to the parity disk as well.

```
UserRead(Disks[]) {
    data[] = RaidRead(Disks[]);
    if(raid read failed)
        Declare double failure and return;
    else
        CheckValid(Disks[], data[]);
}
RaidRead(Disks[]) {
    for(x = 0 to num(Disks[])) {
        data[x] = DiskRead(Disks[x]);
        if(disk read failed) { // LSE
            data[x] = Reconstruct(Disks[x]);
            if(reconstruct failed) { // another LSE
                return FAILURE;
            }
        }
    }
    return data[];
}
Reconstruct(BadDisk) {
    for(x = 0 to num(AllDisks[])) {
        if(Disks[x] is not BadDisk)
            data[x] = DiskRead(Disks[x]);
        else
            data[x] = DiskRead(ParityDisk);
        if(disk read fails) // LSE
            return FAILURE;
    }
    new_data = Parity(data[x]);
    DiskWrite(Disks[x], new_data);
    return new_data;
}
UserWrite(Disks[], data[]) {
    if(Additive parity cost is lower for num(Disks[])) {
        other_disk_data[] = RaidRead(AllDisks[] - Disks[]);
        if(raid read failed)
            Declare double failure and return;
        parity_data = Parity(data[] + other_disk_data[]);
    }
    else{ // subtractive parity
        old_data[] = RaidRead(Disks[] + ParityDisk);
        if(raid read failed)
            Declare double failure and return;
        parity_data = Parity(data[] + old_data[]);
    }
    for(x = 0 to num(Disks[])) {
        DiskWrite(Disks[x], data[x]);
    }
    DiskWrite(ParityDisk, parity_data)
    return SUCCESS;
}
```

Figure 1: **Model of Bare-bones RAID.** *The figure shows the model of bare-bones RAID specified using the primitives* **DiskRead**, **DiskWrite**, **ParityCalc**, *and* **CheckValid** *provided by the model checker.* **CheckValid** *is called when returning data to the user and the model checker verifies if the data is actually valid.*

Let us now consider the state transitions that lead to corrupt data being returned to the user. We retain the names of states involved in these transitions for other data protection schemes as well, since the role they play is similar across schemes.

Any of the errors, lost write, torn write, misdirected write, or corruption to data disk X when in `clean` state, places the model in state `Disk`$_X$ `Error`. In this state, data disk X contains wrong data and the (correct) parity on the stripe is therefore inconsistent with the data disks. A user read to data disk X will now return corrupt data to the user (`Corrupt Data`), simply because there is no means of verifying that the data is valid. If a user write to disks other than data disk X triggers additive parity calculation ($\mathrm{W_{ADD}}(!\mathrm{X})$), the corrupt data in data disk X is used for parity calculation, thereby corrupting the parity disk as well. In this scenario, both data disk X and the parity disk contain corrupt data, but they are consistent. We term this process of propagating incorrect data to the parity disk during additive parity calculation as *parity pollution* and it corresponds to the state `Polluted Parity`. Parity pollution does not impact the probability of data loss or corruption in this case since bare-bones RAID does not detect any form of corruption. However, as we shall see, parity pollution causes problems for many other protection schemes.

When in state `Disk`$_X$ `Error`, if a user write involving data disk X leads to subtractive parity calculation ($\mathrm{W_{SUB}}(\mathrm{X}+)$), the corrupt data in data disk X is used for the parity calculation. Therefore, the new parity generated is corrupt (and also inconsistent with the data disks). However, since data disk X is being written, data disk X is no longer corrupt. This state is named as `Parity Error` in the state machine. We see that the same state can be reached from `clean` state when an error occurs for the parity disk. This state does not lead to further data loss or corruption in the absence of a second error (if a second error is detected on one of the data disks, the corruption will be propagated to that disk as well). Thus, we see that, bare-bones RAID protects only against latent sector errors and not other errors.

4.2 Data Scrubbing

In this scheme, we add data scrubbing to the bare-bones RAID protection scheme. Data scrubbing is an extended version of disk media scrubbing [22, 33]. Data scrubs read all disk blocks that form the stripe and reconstruct the data if an error is detected. The scrub also recomputes the parity of the data blocks and compares it with the parity stored on the parity disk, thereby detecting any inconsistencies [3]. Thus, the scrubbing mechanism can convert the RAID recovery mechanism into an error detection technique. Note that if an inconsistency is detected,

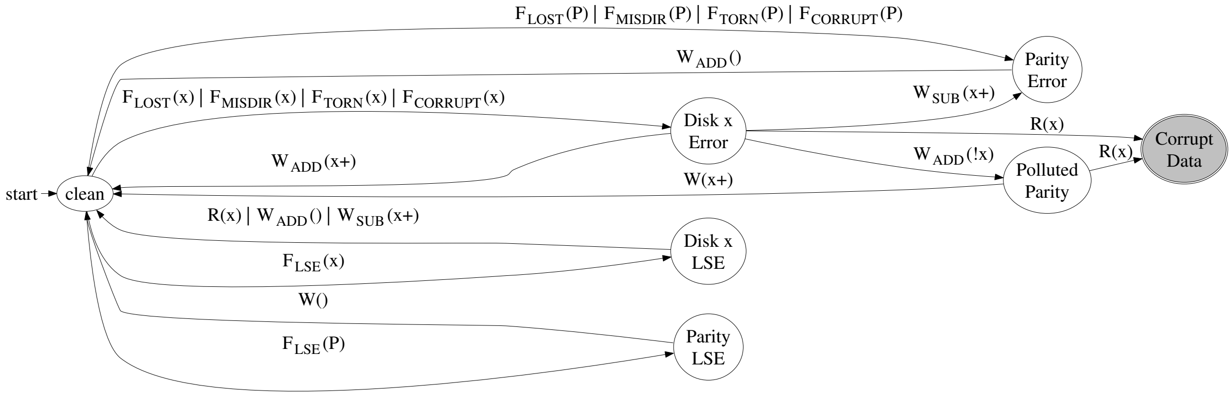

Figure 2: **State Machine for Bare-bones RAID.**

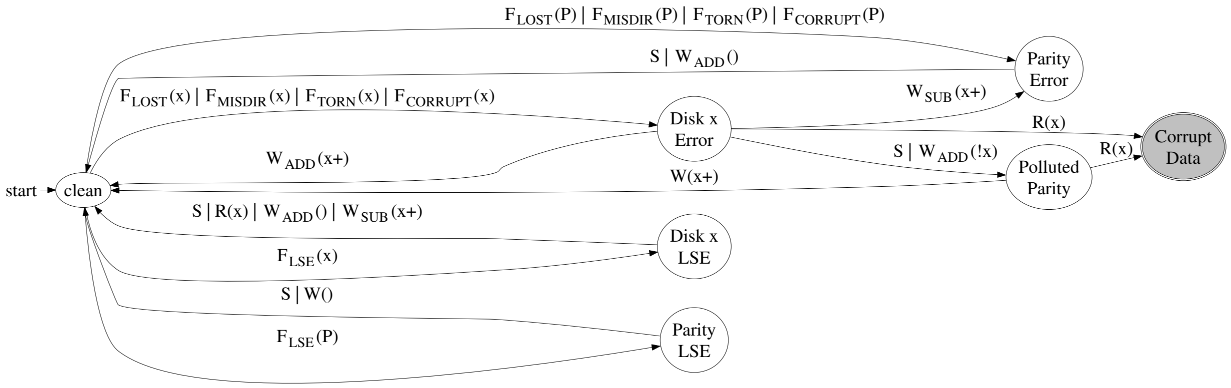

Figure 3: **RAID with scrubbing.**

bare-bones RAID does not offer a method to resolve it. The scrub should fix the inconsistency (by recomputing the contents of the parity disk) because inconsistent data and parity lead to data corruption if a second failure occurs and reconstruction is performed. In the rest of the section, when we refer to data scrubbing, we also imply that the scheme fixes parity inconsistencies.

When the model checker is used to examine this model and only one error is injected, we obtain the state machine shown in Figure 3. We see that the state machine is very similar to that of bare-bones RAID, except that some edges include S. One such edge is the transition from the state `Disk`$_X$ `Error`, where data in data disk X is wrong, to `Polluted Parity`, where both the data and parity are wrong, but consistently so. This transition during a scrub is easily explained – in `Disk`$_X$ `Error`, the scrub detects a mismatch between data and parity and updates the parity to match the data moving the model to state `Polluted Parity`. We see that the addition of the scrub has not improved protection when only one error is injected; scrubs are intended to lower the chances of double failures, not of loss from single errors. In fact, we shall see later that the tendency of scrubs to pollute parity increases the chances of data loss when only one error occurs.

4.3 Checksums

Checksumming techniques have been used in numerous systems over the years to detect data corruption. Some systems store the checksum along with the data that it protects [4, 14, 37], while other systems store the checksum on the access path to the data [35, 36]. We will explore both alternatives. We also distinguish between the schemes that store per sector checksums [4, 14] and those that use per-block checksums [37].

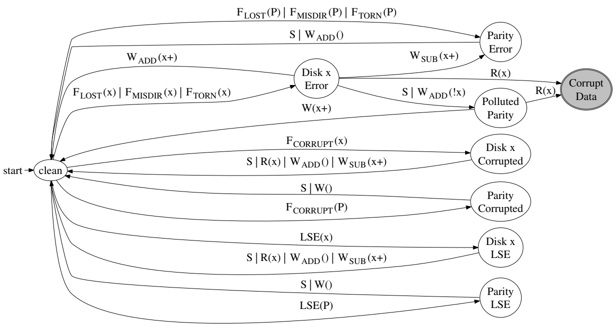

Figure 4: **Sector checksums + RAID and scrubbing.**

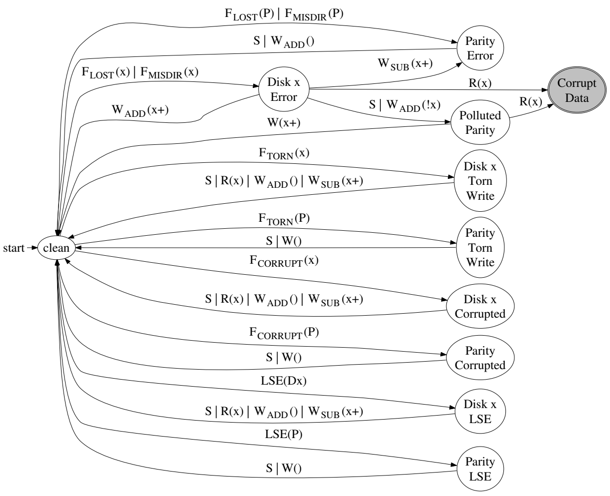

Figure 5: **Block checksums + RAID and scrubbing.**

Sector checksums: Figure 4 shows the state machine obtained for sector-level checksum protection. The obvious change from the previous state machines is the addition of two new states `Disk`$_X$ `Corrupted` and `Parity Corrupted`. The model transitions to these states from the `clean` state when a corruption occurs to data disk X or the parity disk respectively. The use of sector checksums enables the detection of these corruptions whenever the corrupt block is read (including scrubs), thus initiating reconstruction and thereby returning the model to `clean` state. However, the use of sector checksums does not protect against torn writes, lost writes, and misdirected writes. For example, torn writes update a single sector, but not the rest of the block. The checksum for all sectors is therefore consistent with the data in that sector. Therefore, sector checksums do not detect these scenarios ($\mathrm{R}(\mathrm{X})$ from `Disk`$_X$ `Error` leads to `Corrupt Data`).

Block checksums: The goal of block checksums is to ensure that a disk block is one consistent unit, unlike with sector checksums. Figure 5 shows the state machine obtained for block-level checksum protection. Again, the addition of new states that do not lead to `Corrupt Data` signifies an improvement in the protection. The new states added correspond to torn writes. Unlike sector-level protection, block-level protection can detect torn writes (detection denoted by transitions from states `Disk`$_X$ `Torn` and `Parity Torn` to `clean`) in exactly the same manner as detecting corruption. However, we see that corrupt data could still be returned to the user. A lost write or a misdirected write transitions the model from the `clean` state to `Disk`$_X$ `Error`. When a lost write occurs, the disk block retains data and checksum written on a previous occasion. The data and checksum are therefore consistent. Hence, the model does not detect that the data on disk is wrong. A read to data disk X now returns corrupt data to the user. The scenario is similar for misdirected writes as well.

Parental checksums: A third option for checksumming is to store the checksum of the disk block in a parent block that is accessed first during user reads (*e.g.*, an inode of a file is read before its data block). Parental checksums can thus be used to verify data during all user reads, but not for other operations. Figure 6 shows the state machine for this scheme. We notice many changes to the state machine as compared to block checksums. First, we see that the states successfully handled by block checksums (such as $\mathrm{F}_{\mathrm{TORN}}(\mathrm{X})$) do not exist. Instead, the transitions that led from `clean` to those states now place the model in `Disk`$_X$ `Error`. Second, none of the states return corrupt data to the user. Instead, a new node called `Data Loss` has been added. This change signifies that the model detects a double failure and reports data loss. Third, the only transition to `Data Loss` is due to a read of data disk X when in the `Polluted Parity` state. Thus, parity pollution now leads to data loss. As before, the causes of parity pollution are data scrubs or additive parity calculations (transitions S or $\mathrm{W}_{\mathrm{ADD}}(!\mathrm{X})$ lead from `Disk`$_X$ `Error` to `Polluted Parity`). Figure 7 presents a pictorial view of the transitions from clean state to parity pollution and data loss. At the root of the problem is the fact that parental checksums can be verified only for user reads, not other disk reads. Any protection technique that does not co-operate with RAID, allows parity recalculation to use bad data, causing irreversible data loss.

Of the three checksums techniques evaluated, we find that block checksumming has the fewest number of transitions to data loss or corruption. Therefore, we use block checksums as the starting point for adding further protection techniques.

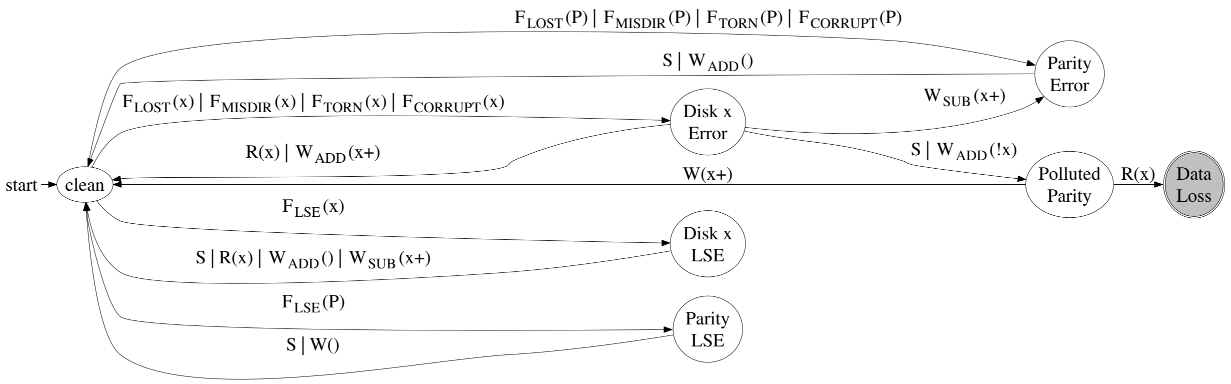

Figure 6: **Parental checksums + RAID and scrubbing.**

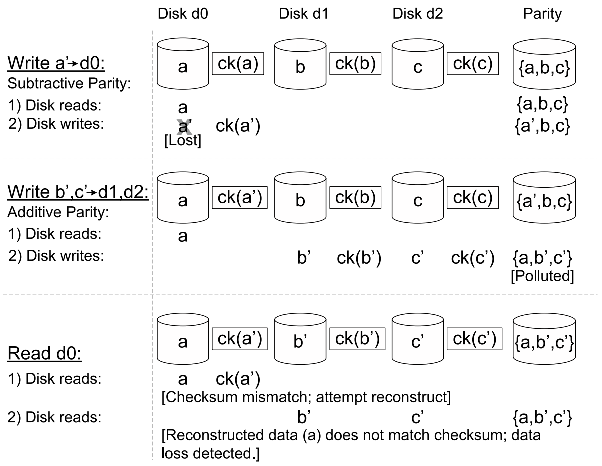

Figure 7: **Parity Pollution Sequence.** *This figure shows a sequence of operations, along with intermediate RAID states, that lead to parity pollution and subsequent data loss. Each horizontal set of disks (Data disks d0, d1 and d2 and Parity disk) form the RAID stripe. The contents of the disk blocks are shown inside the disks. a, b etc. are data values, and $\{a, b\}$ denotes the parity of values a and b. The protection scheme used is parental checksums. Checksums are shown next to the corresponding data disks. At each RAID state, user read or write operations cause corresponding disk reads and writes, resulting in the next state. The first write to disk d0 is lost, while the checksum and parity are successfully updated. Next, a user write to disks d1 and d2 uses the bad data in disk d0 to calculate parity, thereby causing parity pollution. A subsequent user read to disk d0 detects a checksum mismatch, but recovery is not possible since parity is polluted.*

4.4 Write-Verify

One primary problem with block checksums is that lost writes are not detected. Lost writes are particularly difficult to handle. If the checksum is stored along with the data and both are written as part of the same disk request, they are both lost, leaving the old data and checksum intact and valid. On later reads to disk block, checksum verification compares the old data and old checksum which are consistent, thereby not detecting the lost write.

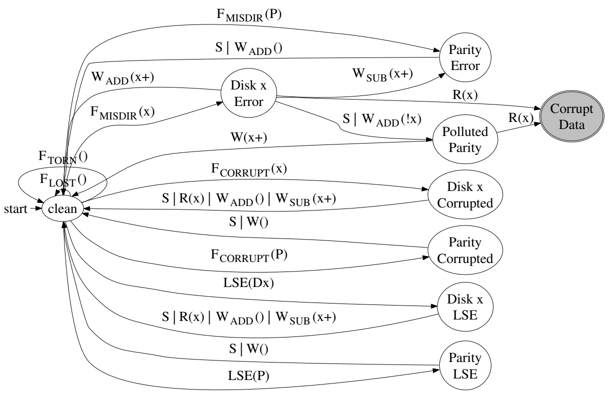

Figure 8: **Write-verify + block checksums, RAID and scrubbing.**

One simple method to fix this problem is to ensure that writes are not lost in the first place. Some storage systems perform write-verify [18, 37] (also called read-after-write verify) for this purpose. This technique reads the disk block back after it is written, and uses the data contents in memory to verify that the write has indeed completed without errors.

Figure 8 shows the state machine for write-verify with block checksums. Comparing this figure against Figure 5, we notice two differences: First, the states representing torn data or parity do not exist anymore. Second, the transitions $\mathrm{F_{TORN}(X)}$, $\mathrm{F_{TORN}(P)}$, $\mathrm{F_{LOST}(X)}$, and $\mathrm{F_{LOST}(P)}$ are now from `clean` to itself, instead of to

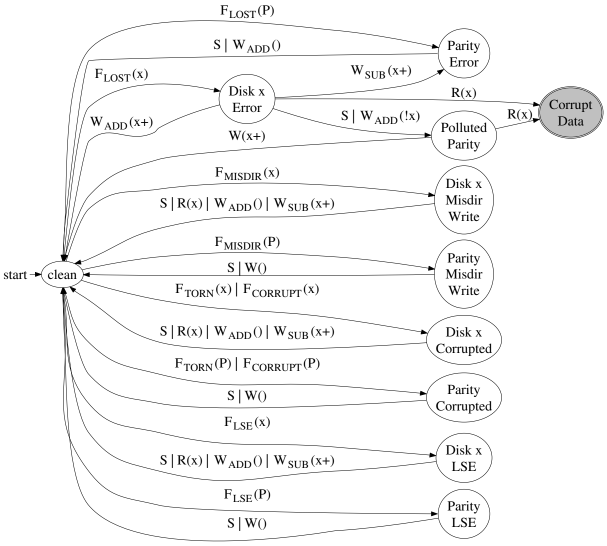

Figure 9: **Physical identity + RAID, scrubbing, and block checksums.**

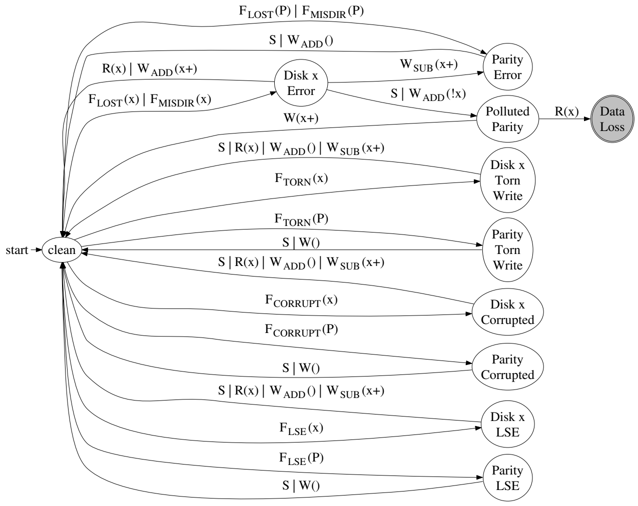

Figure 10: **Logical identity + RAID, scrubbing and block checksums.**

other states (self-loops shown for readability). Write-verify detects lost writes and torn writes as and when they occur, keeping the RAID stripe in clean state.

Unfortunately, write-verify has two negatives. First, it does not protect against misdirected writes. When a misdirected write occurs, Write-verify would detect that the original target of the write suffered a lost write, and therefore simply reissue the write. However, the victim of the misdirected write is left consistent with consistent checksums but wrong data. A later user read to the victim thus returns corrupt data to the user. Second, although write-verify improves data protection, every disk write now incurs a disk read as well, possibly leading to a huge loss in performance.

4.5 Identity

A different approach that is used to solve the problem of lost or misdirected writes without the huge performance penalty of write-verify is the use of identity information.

Different forms of identifying data (also called self-describing data) can be stored along with data blocks. An identity may be in one of two forms: (a) physical identity, which typically consists of the disk number and the disk block (or sector) number to which the data is written [4], and (b) logical identity, which is typically an inode number and offset within the file [31, 37].

• **Physical identity**: Figure 9 shows the state machine obtained when physical identity information is used in combination with block checksums. Compared to previous state machines, we see that there are two new states corresponding to misdirected writes, `Misdirected Data` and `Misdirected Parity`. These states are detected by the model when the disk block is read for any reason (scrub, user read, or parity calculation) since even non-user operations like scrub can verify physical identity. Thus, physical identity is a step towards mitigating parity pollution. However, parity pollution still occurs in state transitions involving lost writes. If a lost write occurs, the disk block contains the old data, which would still have the correct physical block number. Therefore, physical identity cannot protect against lost writes, leading to corrupt data being returned to the user.

• **Logical identity**: The logical identity of disk blocks is defined by the block's parent and can therefore be verified only during user reads. Figure 10 shows the state machine obtained when logical identity protection is used in combination with block checksums. Unlike physical identity, misdirected writes do not cause new states to be created for logical identity. Both lost and misdirected writes place the model in the $\texttt{Disk}_X$ `Error` state. At this point, parity pollution due to scrubs and user writes moves the system to the `Polluted Parity` state since logical identity can be verified only on user reads, thus causing data loss. Thus, logical identity works in similar fashion to parental checksums: (i) in both cases, there is a check that uses data from outside the block being protected, and (ii) in both cases, corrupt data is not returned to the user and instead, data loss is detected.

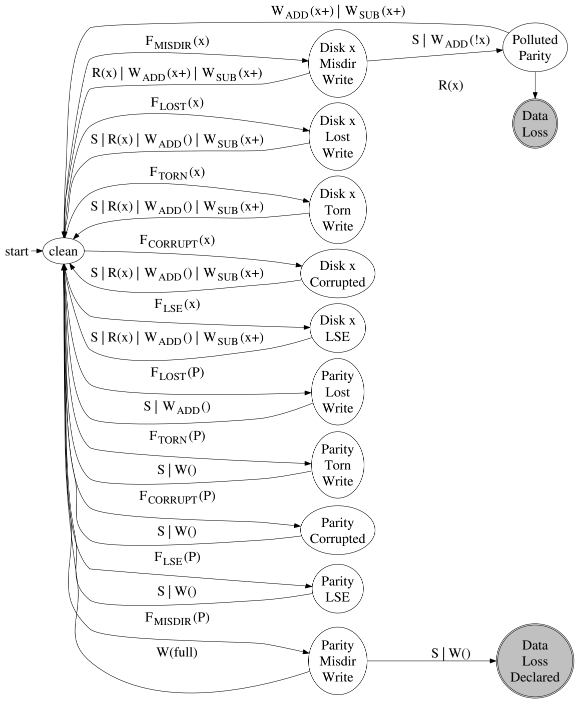

Figure 11: **Version mirroring + Logical identity, block checksums, RAID, and scrubbing.**

4.6 Version Mirroring

The use of identity information (both physical and logical) does not protect data from exactly one scenario – parity pollution after a lost write. Version mirroring can be used to detect lost writes during scrubs and parity calculation. Herein, each data block that belongs to the RAID stripe contains a version number. This version number is incremented with every write to the block. The parity block contains a list of version numbers of all of the data blocks that it protects. Whenever a data block is read, its version number is compared to the corresponding version number stored in the parity block. If a mismatch occurs, the newer block will have a higher version number, and can used to reconstruct the other data block.

Note that when this approach is employed during user reads, each disk block read would now incur an additional read of the parity block. To avoid this performance penalty, version numbers can be used in conjunction with logical identity. Thus, logical identity is verified during file system reads, while version numbers are verified for parity re-calculation reads and disk scrubbing. This approach incurs an extra disk read of the parity block only when additive parity calculation is performed.

A primitive form of version mirroring has been used in real systems: Dell Powervault storage arrays [14] use a 1-bit version number called a "write stamp". However, since the length of the version number is restricted to 1-bit, it can only be used to *detect* a mismatch between data and parity (which we already can achieve through parity recompute and compare). It does not provide the power to identify the wrong data (which would enable recovery). This example illustrates that the bit-length of version numbers limits the number of errors that can be detected and recovered from.

Figure 11 shows the state machine obtained when version mirroring is added to logical identity protection. We find that there are now states corresponding to lost writes (`Lost Data` and `Lost Parity`) for which all transitions lead to `clean`. However, `Data Loss` could still occur, and in addition, `Data Loss Declared` could occur as well. The only error that causes state transitions to any of these nodes is a misdirected write.

A misdirected write to data disk X places the model in `Misdirected Data`. Now, an additive parity calculation that uses data disk X will compare the version number in data disk X against the one in the parity disk. The misdirected write could have written a disk block with a higher version number than the victim. Thus, the model trusts the wrong data disk X and pollutes parity. A subsequent read to data disk X uses logical identity to detect the error, but the parity has already been polluted.

A misdirected write to the parity disk causes problems as well. Interestingly, none of the protection schemes so far face this problem. The sequence of state transitions leading to `Data Loss Declared` occurs in following fashion. A misdirected write to the parity disk places new version numbers in the entire list of version numbers on the disk. When any data disk's version number is compared against its corresponding version number on this list (during a write or scrub), if the parity's (wrong) versions numbers are higher, reconstruction is initiated. Reconstruction will detect that none of the version numbers of the data disks match the version numbers stored on the parity disk. In this scenario, a multi-disk error is detected and the model declares data loss. This state is different from `Data Loss`, since this scenario is a false positive while the other has actual data loss.

The occurrence of the `Data Loss Declared` state indicates that the policy used when multiple version numbers mismatch during reconstruction is faulty. It is indeed possible to have a policy that fixes parity instead of data on a multiple version number mismatch. The use of a model checker thus enables identification of policy faults as well.

We know from the previous subsection that physical identity protects against misdirected writes. Therefore, if physical identity is added to version mirroring and logical identity, we could potentially eliminate all problem nodes. Figure 12 shows the state machine generated for this protection scheme. We see that none of the state tran-

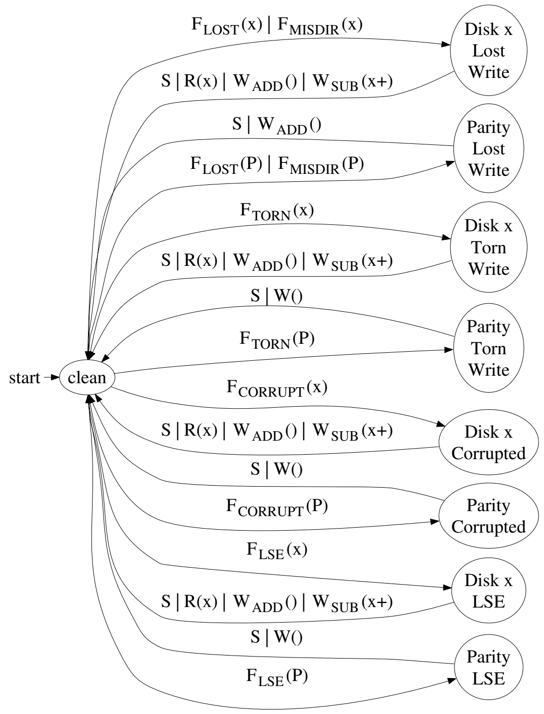

Figure 12: **Version mirroring + Logical and physical identity, block checksums, RAID and scrubbing.**

sitions lead to data loss or data corruption. The advantage of using physical identity is that the physical identity can be verified (detecting any misdirected write) before comparing version numbers. Thus, we have identified a scheme that eliminates data loss or corruption due to a realistic range of disk errors; the scheme includes version mirroring, physical and logical identity, block checksums, and RAID.

4.7 Discussion

The analysis of multiple schemes has helped identify the following key data protection issues.

• **Parity pollution**: We believe that any parity-based system that re-uses existing data to compute parity is potentially susceptible to data loss due to disk errors, in particular lost and misdirected writes. In the absence of techniques to perfectly verify the integrity of existing disk blocks used for recomputing the parity, disk scrubbing and partial-stripe writes can cause parity pollution, where the parity no longer reflects valid data.

In this context, it would be interesting to apply model checking to understand schemes with double parity [7, 13]. Another interesting scheme that could be analyzed is one with RAID-Z [8] protection (instead of RAID-4 or RAID-5), where only full-stripe writes are performed and data is protected with parental checksums.

• **Parental protection**: Verifying the contents of a disk block against any value – either identity or checksum, written using a separate request and stored in a different disk location, is an excellent method to detect errors that are more difficult to handle. However, in the absence of techniques such as version mirroring, schemes that protect data by placing checksum or identity protections on the access path should use the same access path for disk scrubbing, parity calculation, and reconstructing data. Note that this approach could slow down these processes significantly, especially when the RAID is close to full space utilization.

• **Mirroring**: Mirroring of any piece of data, provides a distinct advantage: one can verify the correctness of data through comparison without interference from other data items (as in the case of parity). Version mirroring utilizes this advantage in conjunction with crucial knowledge about the items that are mirrored – the higher value is more recent.

• **Physical identity**: Physical identity, like block checksums, is extremely useful since it is knowledge available at the RAID-level. We see that this knowledge is important for perfect data protection.

• **Recovery-integrity co-design**: Finally, it is vital to integrate data integrity with RAID recovery, and do so by exhaustively exploring all possible scenarios that could occur when the protection techniques are composed.

Thus, a model checking approach is very useful in deconstructing the exact protection offered by a protection scheme, thereby also identifying important data protection issues. We believe that such an exhaustive approach would prove even more important in evaluating protections against double errors.

5 Probability of Loss or Corruption

One benefit of using a model checker is that we can assign probabilities to various state transitions in the state machine produced, and easily generate approximate probabilities for data loss or corruption. These probabilities help compare the different schemes quantitatively.

We use the data for nearline disks in Section 2 to derive per-year probabilities for the occurrence of the different errors. For instance, the probability of occurrence of $\mathrm{F_{LSE}}$ (a latent sector error) for one disk is 0.1. The data does not distinguish between corruption and torn writes; therefore, we assume an equal probability of occurrence of $\mathrm{F_{CORRUPT}}$ and $\mathrm{F_{TORN}}$ (0.0022). We derive the probabilities for $\mathrm{F_{LOST}}$ and $\mathrm{F_{MISDIR}}$ based on the assumptions in Section 3.2 as 0.0003 and $1.88e-5$ respectively.

We also compute the probability for each operation to be the first to encounter the stripe with an existing er-

RAID	Scrub	Sector Checksum	Block Checksum	Parent Checksum	Write-Verify	Physical ID	Logical ID	Version Mirror	Chance of Data Loss
√									0.602%
√	√								0.602%
√	√	√							0.322%
√	√		√						0.041%
√	√			√					*0.486%
√				√					*0.153%
√	√		√		√				0.002%
√	√		√			√			0.038%
√	√		√				√		*0.033%
√			√				√		*0.010%
√	√		√			√	√		*0.031%
√			√			√	√		*0.010%
√	√		√				√	√	*0.004%
√			√				√	√	*0.002%
√	√		√			√	√	√	0.000%

Table 3: **Probability of Loss or Corruption.** *The table provides an approximate probability of at least 1 data loss event and of corrupt data being returned to the user at least once, when each of the protection schemes is used for storing data. It is assumed that the storage system uses 4 data disks, and 1 parity disk. A (*) indicates that the data loss is detectable given the particular scheme (and hence can be turned into unavailability, depending on system implementation).*

ror. For this purpose, we utilize the distribution of how often different requests detect corruption in our study study [3]. The distribution is as follows. P(User read): 0.2, P(User write): 0.2, P(Scrub): 0.6. We assume that partial stripe writes of varying width are equally likely.

Note that while we attempt to use as realistic probability numbers as possible, the goal is not to provide precise data loss probabilities, but to illustrate the advantage of using a model checker, and discuss potential trade-offs between different protection schemes.

Table 3 provides approximate probabilities of data loss derived from the state machines produced by the model checker. We consider a 4 data disk, 1 parity disk RAID configuration for all of the protection schemes for calculating probabilities. This table enables simple comparisons of the different protection schemes. We can see that generally, enabling protections causes an expected decrease in the chance of data loss. The use of version mirroring with logical and physical identity, block checksums and RAID produces a scheme with a theoretical chance of data loss or corruption as 0. The data in the table illustrates the following trade-offs between protection schemes:

Scrub vs. No scrub: Systems employ scrubbing to detect and fix errors and inconsistencies in order to reduce the chances of double failures. However, our analysis in the previous section shows that scrubs could potentially cause data loss due to parity pollution. The data in the table shows that it is indeed the case. In fact, since scrubs have a higher probability of encountering errors, the probability of data loss is significantly higher with scrubs than without. For example, using parental checksums with scrubs causes data loss with a probability 0.00486, while using parental checksums without scrubs causes data loss with a 3 times lesser probability 0.00153.

Data loss vs. Corrupt data: Comparing the different protection schemes, we see that some schemes cause data loss whereas others return corrupt data to the user. Interestingly, we also see that the probability of data loss is higher than the probability of corrupt data. For example, using parental checksums (with RAID and scrubbing) causes data loss with a probability 0.00486, while using block checksums causes corrupt data to be returned with a an order of magnitude lesser probability 0.00041. Thus, while in general it is better to detect corruption and incur data loss than to return corrupt data, the answer may not be obvious when the probability of loss is much higher.

If the precise probability distributions of the underlying errors, and read, write, and scrub relative frequencies are known, techniques like Monte-Carlo simulation can be used to generate actual probability estimates that take multiple errors into consideration [15].

6 Related Work

Many research efforts have explored reliability modeling for RAID-based storage systems, right from when the case was made for RAID storage [29]. Most initial efforts focus on complete disk failures [10, 11, 26, 27]. For example, Burkhard and Menon [10] use Markov models to estimate the reliability provided by multiple check (parity) disks in a RAID group.

More recent research has explored the impact of partial disk failures, such as latent sector errors. Disk scrubbing [22] has been used for many years for proactive detection of latent errors, thus reducing the probability of double failures. Schwarz *et al.* use statistical models to analyze the fault tolerance provided by different options for disk scrubbing in archival storage systems [33]. Elerath and Pecht use Monte Carlo simulation to explore RAID reliability, considering different distributions for disk failures, latent errors, disk scrubbing, and time taken for RAID reconstruction [15]. Most of these research efforts compute the reliability of RAID systems assuming that errors are detected and fixed when encountered (say through scrubbing), while we examine the design of the protections that provide such an assurance.

Sivathanu *et al.* provide a qualitative discussion of the assurances provided by various redundancy techniques [34]. We show that when multiple techniques are used in combination, a more exhaustive exploration of such assurances is essential.

Most related to our work is simultaneous research by Belluomini et al. [5]. They describe how undetected disk errors such as silent data corruptions and lost writes could potentially lead to a RAID returning corrupt data to the user. They explore a general solution space involving addition of an appendix with some extra information to each disk block or the parity block. In comparison, our effort is a detailed analysis of the exact protection offered by each type of extra information like block checksums, physical identity, etc.

Research efforts have also applied fault injection, instead of modeling, as a means to quantify the reliability and availability of RAID storage. Brown and Patterson [9] use benchmarks and fault injection to measure the availability of RAID.

Other research efforts that have leveraged model checking ideas to understand the reliability properties of actual operating system and storage system code [32, 39, 40]. For example, Yang *et al.* use model checking to identify bugs in file system code [40], and later they adapt model checking ideas to find bugs in many different file system and RAID implementations [39]. Model checking has also been used to study security protocols [32].

7 Conclusion

We have presented a formal approach to analyzing the design of data protection strategies. Whereas earlier designs were simple to verify by inspection (*e.g.*, a parity disk successfully adds protection against full-disk failure), modern systems employ a host of techniques, and their interactions are subtle.

With our approach, we have shown that a variety of approaches found in past and current systems are successful at detecting a variety of problems but that some interesting corner-case scenarios can lead to data loss or corruption. In particular, we found that the problem of parity pollution can propagate errors from a single (bad) block to other (previously good) blocks, and thus lead to a gap in protection in many schemes. The addition of version mirroring and proper identity information, in addition to standard checksums, parity, and scrubbing, leads to a solution where no single error should (by design) lead to data loss.

In the future, as protection evolves further to cope with the next generation of disk problems, we believe approaches such as ours will be critical. Although model checking implementations is clearly important [40], the first step in building any successful storage system should begin with a correctly-specified design.

Acknowledgments

This material is based upon work supported by the National Science Foundation under the following grants: CCF-0621487, CNS-0509474, CCR-0133456, as well as by generous donations from Network Appliance.

We thank members of the RAID group at NetApp including Atul Goel, Tomislav Grcanac, Rajesh Sundaram, Jim Taylor, and Tom Theaker for their insightful comments on data protection schemes. David Ford, Stephen Harpster, Steve Kleiman, Brian Pawlowski and members of the Advanced Technology Group at NetApp provided excellent feedback on the work. Finally, we would like to thank our shepherd Darrell Long and the anonymous reviewers for their detailed comments that helped improve the paper significantly.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or other institutions.

References

[1] Adaptec, Inc. Adaptec SCSI RAID 2200S. http://www.adaptec.com/en-US/support/raid/scsi_raid/ASR-2200S/, 2007.

[2] L. N. Bairavasundaram, G. R. Goodson, S. Pasupathy, and J. Schindler. An Analysis of Latent Sector Errors in Disk Drives. In *Proceedings of the 2007 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '07)*, San Diego, CA, June 2007.

[3] L. N. Bairavasundaram, G. R. Goodson, B. Schroeder, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. An Analysis of Data Corruption in the Storage Stack. In *Proceedings of the 6th USENIX Symposium on File and Storage Technologies (FAST '08)*, San Jose, California, February 2008.

[4] W. Bartlett and L. Spainhower. Commercial Fault Tolerance: A Tale of Two Systems. *IEEE Transactions on Dependable and Secure Computing*, 1(1):87–96, January 2004.

[5] W. Belluomini, V. Deenadhayalan, J. L. Hafner, and K. Rao. Undetected Disk Errors in RAID Arrays. *To appear in the IBM Journal of Research and Development*.

[6] D. Bitton and J. Gray. Disk shadowing. In *Proceedings of the 14th International Conference on Very Large Data Bases (VLDB 14)*, pages 331–338, Los Angeles, California, August 1988.

[7] M. Blaum, J. Brady, J. Bruck, and J. Menon. EVENODD: An optimal scheme for tolerating double disk failures in RAID architectures. In *Proceedings of the 21st Annual International Symposium on Computer Architecture (ISCA '94)*, pages 245–254, Chicago, Illinois, April 1994.

[8] J. Bonwick. RAID-Z. http://blogs.sun.com/bonwick/entry/raid_z, Nov. 2005.

[9] A. Brown and D. A. Patterson. Towards Availability Benchmarks: A Case Study of Software RAID Systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX '00)*, pages 263–276, San Diego, California, June 2000.

[10] W. Burkhard and J. Menon. Disk Array Storage System Reliability. In *Proceedings of the 23rd International Symposium on Fault-Tolerant Computing (FTCS-23)*, pages 432–441, Toulouse, France, June 1993.

[11] P. Chen and E. K. Lee. Striping in a RAID Level 5 Disk Array. In *Proceedings of the 1995 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '95)*, pages 136–145, Ottawa, Canada, May 1995.

[12] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: High-performance, Reliable Secondary Storage. *ACM Computing Surveys*, 26(2):145–185, June 1994.

[13] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar. Row-Diagonal Parity for Double Disk Failure Correction. In *Proceedings of the 3rd USENIX Symposium on File and Storage Technologies (FAST '04)*, pages 1–14, San Francisco, California, April 2004.

[14] M. H. Darden. Data Integrity: The Dell—EMC Distinction. http://www.dell.com/content/topics/global.aspx/power/en/ps2q02_darden?c=us&cs=555&l=en&s=biz, May 2002.

[15] J. Elerath and M. Pecht. Enhanced Reliability Modeling of RAID Storage Systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN-2007)*, Edinburgh, United Kingdom, June 2007.

[16] Gentoo HOWTO. HOWTO Install on Software RAID. http://gentoo-wiki.com/HOWTO_Gentoo_Install_on_Software_RAID, Sept. 2007.

[17] R. Green. EIDE Controller Flaws Version 24. http://mindprod.com/jgloss/eideflaw.html, February 2005.

[18] Hitachi Data Systems. Hitachi Thunder 9500 V Series with Serial ATA: Revolutionizing Low-cost Archival Storage. www.hds.com/assets/pdf/wp_157_sata.pdf, May 2004.

[19] Hitachi Data Systems. Data Security Solutions. http://www.hds.com/solutions/storage-strategies/data-security/solutions.html, Sept. 2007.

[20] E. M. C. Jr., O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[21] H. H. Kari. *Latent Sector Faults and Reliability of Disk Arrays*. PhD thesis, Helsinki University of Technology, September 1997.

[22] H. H. Kari, H. Saikkonen, and F. Lombardi. Detection of Defective Media in Disks. In *The IEEE International Workshop on Defect and Fault Tolerance in VLSI Systems*, pages 49–55, Venice, Italy, October 1993.

[23] K. Keeton, C. Santos, D. Beyer, J. Chase, and J. Wilkes. Designing for disasters. In *Proceedings of the 3rd USENIX Symposium on File and Storage Technologies (FAST '04)*, San Francisco, California, April 2004.

[24] M. Y. Kim. Synchronized disk interleaving. *IEEE Transactions on Computers*, C-35(11):978–988, November 1986.

[25] L. Lancaster and A. Rowe. Measuring Real World Data Availability. In *Proceedings of the LISA 2001 15th Systems Administration Conference*, pages 93–100, San Diego, California, December 2001.

[26] J. Menon and D. Mattson. Comparison of Sparing Alternatives for Disk Arrays. In *Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA '92)*, pages 318–329, Gold Coast, Australia, May 1992.

[27] C. U. Orji and J. A. Solworth. Doubly Distorted Mirrors. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data (SIGMOD '93)*, Washington, DC, May 1993.

[28] J. Ostergaard and E. Bueso. The Software-RAID HOWTO. http://tldp.org/HOWTO/html_single/Software-RAID-HOWTO/, June 2004.

[29] D. Patterson, G. Gibson, and R. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the 1988 ACM SIGMOD Conference on the Management of Data (SIGMOD '88)*, pages 109–116, Chicago, Illinois, June 1988.

[30] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. IRON File Systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 206–220, Brighton, United Kingdom, October 2005.

[31] D. D. Redell, Y. K. Dalal, T. R. Horsley, H. C. Lauer, W. C. Lynch, P. R. McJones, H. G. Murray, and S. C.Purcell. Pilot: An Operating System for a Personal Computer. *Communications of the ACM*, 23(2):81–92, February 1980.

[32] B. Schwarz, H. Chen, D. Wagner, G. Morrison, J. West, J. Lin, and W. Tu. Model Checking An Entire Linux Distribution for Security Violations. In *Proceedings of the Annual Computer Security Applications Conference*, Tucson, Arizona, Dec. 2005.

[33] T. J. Schwarz, Q. Xin, E. L. Miller, D. D. Long, A. Hospodor, and S. Ng. Disk Scrubbing in Large Archival Storage Systems. In *Proceedings of the 12th Annual Meeting of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, Volendam, Netherlands, October 2004.

[34] G. Sivathanu, C. P. Wright, and E. Zadok. Ensuring Data Integrity in Storage: Techniques and Applications. In *The 1st International Workshop on Storage Security and Survivability (StorageSS '05)*, FairFax County, Virginia, November 2005.

[35] C. A. Stein, J. H. Howard, and M. I. Seltzer. Unifying File System Protection. In *Proceedings of the USENIX Annual Technical Conference (USENIX '01)*, Boston, Massachusetts, June 2001.

[36] Sun Microsystems. ZFS: The last word in file systems. www.sun.com/2004-0914/feature/, 2006.

[37] R. Sundaram. The Private Lives of Disk Drives. http://www.netapp.com/go/techontap/matl/sample/0206tot_resiliency.html, February 2006.

[38] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID Hierarchical Storage System. *ACM Transactions on Computer Systems*, 14(1):108–136, February 1996.

[39] J. Yang, C. Sar, and D. Engler. EXPLODE: A Lightweight, General System for Finding Serious Storage System Errors. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, Seattle, Washington, November 2006.

[40] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using Model Checking to Find Serious File System Errors. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, San Francisco, California, December 2004.