Toward I/O-Efficient Protection Against Silent Data Corruptions in RAID Arrays

Mingqiang Li and Patrick P. C. Lee
Department of Computer Science and Engineering, The Chinese University of Hong Kong
Email: mingqiangli.cn@gmail.com, pclee@cse.cuhk.edu.hk

***Abstract*—Although RAID is a well-known technique to protect data against disk errors, it is vulnerable to silent data corruptions that cannot be detected by disk drives. Existing integrity protection schemes designed for RAID arrays often introduce high I/O overhead. Our key insight is that by properly designing an integrity protection scheme that adapts to the read/write characteristics of storage workloads, the I/O overhead can be significantly mitigated. In view of this, this paper presents a systematic study on I/O-efficient integrity protection against silent data corruptions in RAID arrays. We formalize an integrity checking model, and justify that a large proportion of disk reads can be checked with simpler and more I/O-efficient integrity checking mechanisms. Based on this integrity checking model, we construct two integrity protection schemes that provide complementary performance advantages for storage workloads with different user write sizes. We further propose a quantitative method for choosing between the two schemes in real-world scenarios. Our trace-driven simulation results show that with the appropriate integrity protection scheme, we can reduce the I/O overhead to below 15%.**

***Keywords*-RAID; silent data corruptions; integrity protection schemes; I/O overhead**

I. Introduction

Maintaining data reliability is a key design concern in modern storage systems. RAID (Redundant Array of Independent Disks) [5, 23, 41] has been widely adopted to protect disk-based storage systems from losing data due to disk failures [26, 36] and latent sector errors [2, 35]. It operates by employing redundancy (e.g., using erasure coding [28]) in a disk array to recover any error detected by disk drives. However, disk-based storage systems also suffer from a significant number of *silent data corruptions* [3, 10, 17], which can silently corrupt disk data without being detected by disk drives. Silent data corruptions occur mainly due to firmware or hardware malfunctions in disk drives, and can manifest in different forms (see Section II-B). Traditional RAID is designed to protect against detectable errors such as disk failures and latent sector errors, but cannot detect silent data corruptions. To maintain data reliability, we often integrate additional integrity protection mechanisms into RAID to protect against silent data corruptions.

Adding new protection mechanisms to RAID often implies additional I/O overhead. First, due to the complicated causes and manifestations of silent data corruptions, we need to store integrity metadata separately from each data chunk [10, 19]. This introduces additional disk writes. Also, to avoid missing any silent data corruption, a detection step must be carried out in each disk read. For each data chunk to be read, the detection step also reads the separately stored integrity metadata for checking. This introduces additional disk reads. To address this seemingly inevitable I/O overhead problem, existing approaches either provide different solutions that make trade-offs between error detection capabilities and I/O performance [10], or delay the detection to the upper layer (e.g., the file system layer [4, 29, 32]). The former often implies the degradation of data reliability, while the latter implies additional recovery support from the upper layer. We thus pose the following question: *Can we mitigate the I/O overhead of integrity protection against silent data corruptions, while preserving all necessary detection capabilities and making the detection deployable in the RAID layer?*

Our observation is that the design of an integrity protection scheme should *adapt* to real-world storage workloads. By choosing the appropriate scheme according to the read/write characteristics, we can reduce the I/O overhead to an acceptable level, while still protecting against all types of silent data corruptions in the RAID layer. Existing studies (e.g., [10, 19]) only briefly mention the I/O overhead of integrity protection at best, without any in-depth analysis based on real-world workloads. Thus, a key motivation of this work is to systematically examine the I/O overhead and detection capabilities of different integrity protection schemes, which would be of great practical significance to storage system practitioners. In this paper, we make the following contributions.

First, we present a taxonomy of the existing integrity primitives that form the building blocks of integrity protection schemes. We carefully examine the I/O overhead and detection capability of each integrity primitive. To our knowledge, this is the first systematic study of analyzing and comparing existing integrity primitives.

Second, we formalize an integrity checking model and show that a large proportion of disk reads can be checked with simpler integrity checking mechanisms with lower I/O overhead. This integrity checking model guides us to design I/O-efficient integrity protection schemes for RAID.

Third, we construct two I/O-efficient integrity protection schemes for RAID that can detect and locate all types of silent data corruptions. The first one is a variant of the scheme proposed in [19], while the second one is newly proposed by this work. We further propose a low-overhead protection mechanism tailored for parity chunks. We conduct detailed I/O analysis, and show that the two schemes provide complementary performance advantages. We also propose a simple and feasible quantitative method for choosing between the two schemes for different types of storage workloads.

Finally, we evaluate the computational and I/O overheads of the two schemes we construct. We show that both schemes have low computational overhead that will not make the CPU become the bottleneck. We further conduct trace-driven simulation under various realistic storage workloads [16]. We show that by always choosing the more efficient scheme, the I/O overhead can be kept reasonably low (often below 15% of additional disk I/Os). On the other hand, using an existing integrity protection approach can have up to 43.74% of additional disk I/Os.

The rest of the paper proceeds as follows. In Section II, we present the background details for our study. In Section III, we provide a taxonomy of existing integrity primitives. In Section IV, we formalize the integrity checking model. In Section V, we construct and analyze different integrity protection schemes. In Section VI, we propose a quantitative method of choosing the right integrity protection scheme for a given type of workloads. In Section VII, we evaluate the computational and I/O overheads of the integrity protection schemes as well as the effectiveness of our quantitative method of choosing the right scheme. In Section VIII, we review related work. Finally, in Section IX, we conclude the paper.

II. Preliminaries

In this section, we lay out the background details for our study. We first define the terminologies for RAID arrays. We then formulate the problem of silent data corruptions. Finally, we state our goals and assumptions.

A. RAID

RAID (Redundant Array of Independent Disks) [5, 23, 41] is a well-known technique for protecting data against *disk failures* [26, 36], in which the entire disks are inaccessible, and *latent sector errors* [2, 35], in which some disk sectors are inaccessible. We consider a RAID array composed of n homogeneous disks. Each disk has its storage space logically segmented into a sequence of continuous *chunks* of the same size. Here, the chunk size is often a multiple of sector size, which is typically 512 bytes in commodity disks. Let r be the number of sectors contained in a chunk. We define a *stripe* as a group of n chunks located in the same offset of each disk. Within each stripe, an (n, k) erasure code [28] is often adopted to encode k data chunks to obtain m parity chunks, where $m = n - k$, so as to tolerate the failures of any m chunks. Two representative examples of RAID schemes are RAID-5 with $m = 1$ and RAID-6 with $m = 2$. Figure 1 illustrates a stripe of a RAID-6 system.

We call reads and writes issued by the upper file system layer to the RAID layer to be *user reads* and *user writes*, respectively. To simplify our description, when mentioning a user read/write, we often assume the read/write falls within one stripe. Then, a user read/write can be classified as a *full-stripe* or *partial-stripe* one, which touches all data chunks or only a subset of data chunks in a stripe, respectively. In RAID, user reads are directly mapped to disk reads, yet user writes are more complicated since all parity chunks must be updated accordingly. In a full-stripe user write, parity chunks are computed directly from the written chunks, and no additional disk reads are needed for existing chunks [5]. In contrast, in a partial-stripe write, depending on the number of data chunks touched, one of the following two modes is used to update a parity chunk [5]:

- *RMW (Read-Modify-Write):* It first reads each old parity chunk from disk and all data chunks to be touched by the user write. It then computes the parity delta by a linear combination of the difference value of each touched data chunk, and modifies the parity chunk by adding it with the parity delta. Finally, it writes the new parity chunk to disk.
- *RCW (ReConstruct-Write):* It first reads the data chunks that are not touched by the user write from disk. It then reconstructs each parity chunk from all data chunks in the stripe (including the untouched data chunks that are just read and the newly written chunks). Finally, it writes the new parity chunk to disk.

Consider a user write within a stripe. Let τ be the number of touched data chunks. Then the numbers of disk I/Os incurred in the RMW and RCW modes, denoted by $\mathcal{X}_{RMW}$ and $\mathcal{X}_{RCW}$, respectively, are:

$$\mathcal{X}_{RMW} = 2(\tau + m) \quad \text{and} \quad \mathcal{X}_{RCW} = n. \tag{1}$$

In a small user write where τ is small and $\mathcal{X}_{RMW} \leq \mathcal{X}_{RCW}$, RMW mode is adopted; otherwise in a large user write, RCW mode should be used.

B. Silent Data Corruptions

Besides disk failures and latent sector errors, a RAID array can also suffer from silent data corruptions [3, 10, 17]. Unlike disk failures and latent sector errors, silent data corruptions

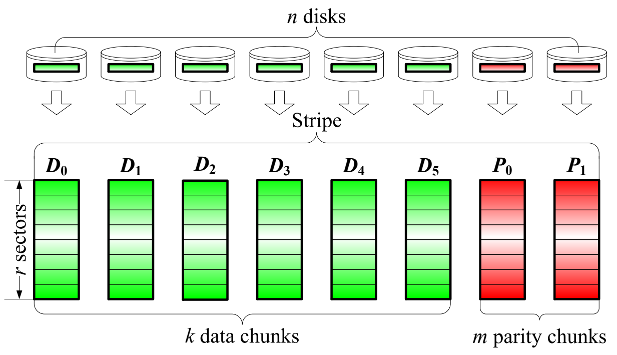
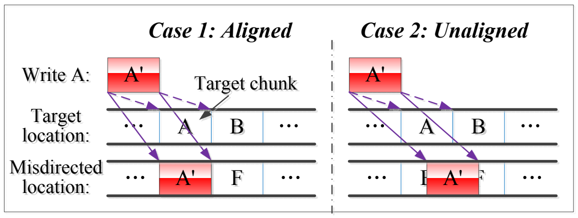
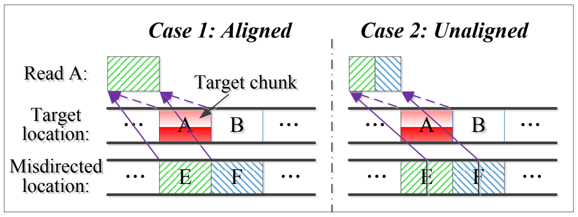


Fig. 1. A stripe of RAID-6 with $n = 8$ disks, in which there are $k = 6$ data chunks and $m = 2$ parity chunks. Assuming that each chunk has a size of 4KB, so it has $r = 8$ sectors of 512 bytes each.



(a) Misdirected writes



(b) Misdirected reads

Fig. 2. Examples of aligned and unaligned misdirected writes and reads.

cannot be detected by disk drives themselves, and thus can corrupt disk data without being notified.

Silent data corruptions can be caused by both disk reads and disk writes. Specifically, there are four representative types of silent data corruptions from disk drives[1] [3, 10, 19, 20, 29]:

- *Lost write:* When a disk write is issued to a chunk, all sectors in the chunk are not updated accordingly, and the chunk becomes stale.
- *Torn write:* When a disk write is issued to a chunk, only a portion of sectors in the chunk are successfully updated, and the chunk contains some stale sectors in the end part.
- *Misdirected write:* When a disk write is issued to a chunk (called *target*), the data is written to a wrong disk location, and one or two neighboring chunks (called *victims*) are corrupted. Specifically, if the misdirected write is *aligned* with chunk boundaries, then a victim chunk is entirely corrupted; if it is *unaligned*, then two neighboring victim chunks are partially corrupted (see Figure 2(a)). Since each stripe has one chunk per disk, the two victim chunks corrupted by an unaligned misdirected write belong to two neighboring stripes. A misdirected write is often due to an off-track error [10], and hence the target chunk becomes stale and will not be (partially) overwritten. Since a stale target chunk has already been treated as a lost write, we only consider a misdirected write from the perspective of a victim chunk.
- *Misdirected Read:* When a disk read is issued to a chunk, the disk head is not correctly positioned. Thus, it reads data from a wrong location that is either aligned or unaligned with chunk boundaries (see Figure 2(b)).

Silent data corruptions can cause stale or corrupted data returned by disk reads without any indication. Without proper protection, they can manifest in different operations:

- *User read:* Obviously, corrupted data is directly propagated to upper layers.
- *User write:* Parity chunks are calculated from existing data chunks returned by disk reads. If one of the data chunks is silently corrupted, the parity chunks are wrongly calculated, leading to parity pollution [19].
- *Data reconstruction:* Reconstruction can be triggered when failed disks or sectors are discovered. A silent data corruption on a surviving data or parity chunk can be propagated to the reconstructed data chunks [10, 20].

Silent data corruptions are more dangerous than disk failures and latent sector errors, both of which make data inaccessible and can be detected when a disk read fails to return data. To protect data against silent data corruptions, we need to add to the RAID layer an *integrity protection scheme*, which often uses additional integrity metadata (e.g., checksums) to detect and locate stale or corrupted data. The detected corruptions can then be recovered through RAID redundancy.

C. Goals and Assumptions

In this work, we design I/O-efficient integrity protection schemes for reinforcing RAID arrays to protect against silent data corruptions. We aim for the following three goals:

(1) All types of silent data corruptions, including lost writes, torn writes, misdirected writes, and misdirected reads, should be detected.
(2) The computational and I/O overheads of generating and storing the integrity metadata should be as low as possible.
(3) The computational and I/O overheads of detecting silent data corruptions should be as low as possible.

We propose to deploy our integrity protection schemes in the RAID layer, which is considered to be the best position to detect and correct silent data corruptions [10]. In this case, RAID redundancy can be leveraged to recover any detected silent data corruption. Nevertheless, motivated by the need of end-to-end data integrity for storage systems, recent studies also propose integrity protection schemes in the upper layers. We discuss them in detail in Section VIII.

For a given RAID scheme with the configuration parameter m, we make the following assumptions in our study:

(1) There is at most one silently corrupted chunk within a stripe[2]. This assumption is often made in the literature [1, 10, 19, 21], since having more than one silently corrupted chunk simultaneously in a stripe is very unlikely in practice [3, 17].
(2) When a stripe contains a silently corrupted chunk, no more than $m-1$ other chunks in the stripe can be simultaneosly failed due to disk failures or latent sector errors. This assumption follows the principle that when silent data corruptions are detected, they should then be recovered via RAID redundancy. We thus do not consider the extreme

[1]Disk drives may also suffer from "*bit rot*" errors, which cause random bit flips. However, such errors can be corrected by the built-in ECC mechanism, or they manifest as latent sector errors if an ECC error occurs. Thus, we do not regard such errors as silent data corruptions in this paper.

[2]Note that an unaligned misdirected write corrupts two chunks, but the chunks belong to different stripes (see Section II-B).

case where a silent data corruption occurs when the RAID array is in critical mode [10].

Our study uses the cases of RAID-5 with $m = 1$ and RAID-6 with $m = 2$ as examples of illustration, although it is applicable for general m.

III. Taxonomy of Integrity Primitives

To cope with silent data corruptions, existing studies propose various *integrity primitives*, which form the building blocks for complete data integrity protection schemes. In this section, we provide a taxonomy study of existing integrity primitives. We discuss their I/O overhead and detection capabilities. Our study provides foundations for constructing integrity protection schemes in Section V.

Here, we only examine the integrity primitives that are deployed in the RAID layer (see Section II-C). Since most integrity primitives include special metadata information alongside with data/parity chunks to detect silent data corruptions, we require that the integrity metadata be generated from the RAID layer and the underlying disk drives.

We point out that although each integrity primitive discussed in this section has been proposed in the literature, the comparison of their detection capabilities remains open. To our knowledge, our work is the first one to present a systematic taxonomy study of integrity primitives.

A. Self-Checksumming

Self-checksumming [3, 19] co-locates a checksum of each data chunk at the chunk end (see Figure 3(a)), such that both the data chunk and its checksum can be read or written together in one disk I/O. It is a *self-checking* primitive, since the integrity checking does not involve any disk read to another chunk. It can detect and locate a corrupted data chunk that is partially updated due to a torn write or unaligned misdirected reads and writes, since the data chunk and its checksum become inconsistent. However, it fails to protect against lost writes and aligned misdirected reads/writes as the data chunk and its checksum, while being consistent, are stale (for lost writes) or unexpectedly overwritten (for aligned misdirected reads/writes).

B. Physical Identity

Physical identity [19] co-locates a physical identity value (e.g., the disk address of the data chunk) of each data chunk at the chunk end (see Figure 3(a)). It is a self-checking primitive just like self-checksumming, and both the data chunk and its co-located physical identity can be read and written together in one disk I/O. It can detect and locate a corrupted data chunk that is overwritten by an aligned misdirected write or has its end part modified by an unaligned misdirected write, as well as a wrong chunk returned by an aligned/unaligned misdirected read. In such cases, the physical identity becomes corrupted or inconsistent with where the data chunk is actually stored. However, the physical identity may remain intact in a lost write, torn write, or an unaligned misdirected write that modifies the front part of a chunk, so the protection fails.

C. Version Mirroring

Version mirroring [6, 7, 19] co-locates a version number at the end of each data chunk and appends a copy of the version number to each of the m parity chunks in the same stripe (see Figure 3(b)). It is a *cross-checking* primitive since the checking involves a disk read to another chunk. Note that the layout is an instance of the *data parity appendix method* [10]. Whenever a parity chunk is updated, its attached version numbers can be updated in the same disk I/O.

In version mirroring, the version number often uses a sequence number individually maintained for each data chunk. In each disk write to a data chunk, the version number is incremented via a read-modify-write operation. In small user writes that use RMW mode, updating the version number does not introduce any additional disk I/O, as it can "free-ride" the disk I/Os for the updates of the data and parity chunks. However, in large user writes that use RCW mode, additional τ disk reads are needed for reading the old version numbers of the touched data chunks.

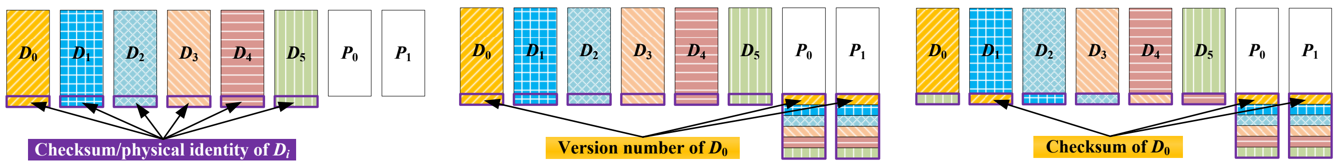
Version mirroring can detect and locate a stale data chunk due to a lost write, by checking if the data chunk has the same up-to-date version number as that attached to one of the parity chunks. However, it cannot detect other types of silent data corruptions, since a version number only records the version of a data chunk but does not check the content of the data chunk.

D. Checksum Mirroring

Checksum mirroring [10] attaches a checksum of each data chunk to the end of the neighboring chunk (called buddy) in the same stripe and also appends a checksum copy to the end of each of the m parity chunks (see Figure 3(c)). The layout of the checksums follows the *buddy parity appendix method* [10]. Like version mirroring, it is a cross-checking primitive, and when a parity chunk is updated, the associated checksums can be updated accordingly in the same disk I/O.

For each data chunk, checksum mirroring attaches checksum copies to $m+1$ other chunks in the same stripe. Thus, if $m-1$ other chunks are failed (as assumed in Section II-C), then there are still two available checksum copies to detect inconsistency and locate a silent data corruption. For example, in Figure 3(c), when P_0 is failed, and if an inconsistency is detected between D_0 and the checksum copy attached to P_1, the checksum copy attached to D_1 can be used to determine whether D_0 or P_1 (including the attached checksum copy) is silently corrupted.

In a partial-stripe write, the checksum updates always involve one additional disk I/O to the buddy of the last touched data chunks, and it applies to both RMW and RCW modes. For example, in a user write that touches the data chunks $D_i, D_{i+1}, \cdots, D_j$ (where $0 \leq i < j \leq k-1$ and $j-i+1 < k$), an additional disk I/O is needed for updating the checksum copy of D_j attached to D_{j+1}. In a full-stripe user write, however, there is no additional I/O since all data chunks in a stripe will be touched. Thus, compared to version mirroring, checksum mirroring has one additional disk I/O in RMW



(a) Self-checksumming/physical identity (b) Version mirroring (c) Checksum mirroring

Fig. 3. Examples of four integrity primitives in RAID-6 (with $n = 8$ disks).

TABLE I

FOUR INTEGRITY PRIMITIVES AND THEIR DETECTION CAPABILITIES FOR DIFFERENT TYPES OF SILENT DATA CORRUPTIONS. WE ALSO CONSTRUCT DIFFERENT INTEGRITY PROTECTION SCHEMES FROM THE PRIMITIVES (SEE SECTION V FOR DETAILS).

Integrity Primitives		**Detection Capabilities for Different Types of Silent Data Corruptions**							**Integrity Protection Schemes**		
		Lost write	Torn write	Misdirected write			Misdirected read		PURE (◇)	HYBRID-1 (♣)	HYBRID-2 (♠)
				Aligned	Unaligned		Aligned	Un-aligned			
					Front-part	End-part					
Self-checksumming	*Self-checking*		✓		✓	✓		✓		♣	♠
Physical identity				✓		✓	✓	✓		♣	♠
Version mirroring	*Cross-checking*	✓								♣	
Checksum mirroring		✓	✓	✓	✓	✓	✓	✓	◇		♠

mode, but can have fewer disk I/Os in RCW mode depending on the number of data chunks (i.e., τ) being updated. We can see that versioning mirroring and checksum mirroring provide complementary performance advantages in different user write modes.

Checksum mirroring can detect and locate all types of silent data corruptions, due to the adoption of checksums for data chunks as well as the separation of data chunks and their checksum copies. Thus, it provides complete integrity protection as opposed to previous integrity primitives.

E. Summary

For comparison, we summarize the detection capabilities of the above four integrity primitives in Table I. An open issue is how to integrate the integrity primitives into a single integrity protection scheme that can detect and locate all types of silent data corruptions while incurring low I/O overhead. We address this issue in the following sections.

IV. INTEGRITY CHECKING MODEL

The detection of silent data corruptions should be carried out in each disk read, which in turn determines the correctness of different operations including user reads, user writes, and data reconstruction (see Section II-B). We present an integrity checking model that formalizes how we detect silent data corruptions in disk reads. We also discuss the implementation details in practice.

A. Formulation

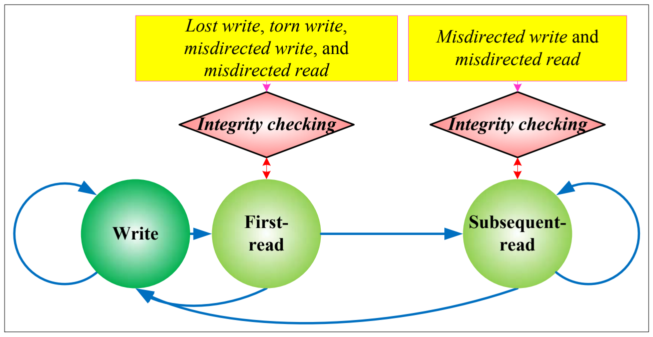
Our integrity checking model mainly classifies the disk reads to a given chunk into two types: (1) the *first-read* after each write to the chunk, and (2) the *subsequent-reads* after the first-read to the chunk. Our observation is that the first-read and subsequent-reads suffer from different types of silent data corruptions, and this motivates us to apply integrity protection for them differently.

Figure 4 depicts how silent data corruptions affect the first-read and subsequent-reads. Specifically, the first-read sees the following types of silent data corruptions: (1) a lost write or a torn write caused by the last disk write to the same chunk, (2) a misdirected write caused by a disk write to a different chunk, or (3) a misdirected read caused by the current disk read. Thus, the first-read can see *all* types of silent data corruptions. On the other hand, a subsequent-read sees the following two types of silent data corruptions: (1) a misdirected write caused by a disk write to a different chunk, or (2) a misdirected read caused by the current read.

This model reveals an important fact that *while the first-read sees all types of silent data corruptions, each subsequent-read only sees a subset of types of silent data corruptions*. The reason is that both the lost write and the torn write can be detected in the first-read, so they do not need to be considered again in subsequent-reads. This fact will guide us to adopt a simpler and accordingly lower-overhead integrity checking mechanism for each subsequent-read when we design our integrity protection schemes.

B. Discussion

To distinguish between the first-read and subsequent-reads, we can employ a bitmap to record the read status of each chunk in a RAID array. We maintain the bitmap as follows. Initially, each bit in the bitmap is set to '0' to indicate that the coming read is the first-read. After the first-read, the corresponding bit is changed to '1' to indicate that the following reads are subsequent-reads. The bit remains unchanged for any subsequent-read until a disk write to the chunk happens. After each disk write, the bit is reset to '0'.



Lost write, torn write, misdirected write, and *misdirected read*
Misdirected write and *misdirected read*
Integrity checking
Integrity checking
Write
First-read
Subsequent-read

Fig. 4. Integrity checking model that describes how silent data corruptions affect the first-read and subsequent-reads.

In practice, the bitmap may be too large to be entirely cached in the RAID controller. To elaborate, let S_{chunk} be the chunk size, and let S_{RAID} be the total capacity of the RAID array. Thus, the bitmap size is S_{RAID}/S_{chunk} bits. Take S_{chunk} = 4KB and S_{RAID} = 8TB for example. Then the bitmap size is 256MB, which makes caching the entire bitmap in the RAID controller infeasible. One plausible solution is to divide the RAID storage space into smaller zones and maintain a much smaller sub-bitmap for each RAID zone. The sub-bitmaps are cached in the RAID controller when the corresponding zones are recently accessed. Due to the strong spatial and temporal localities of practical storage workloads [11, 16, 30, 34], we expect that only a small number of sub-bitmaps need to be cached.

V. Integrity Protection Schemes for RAID

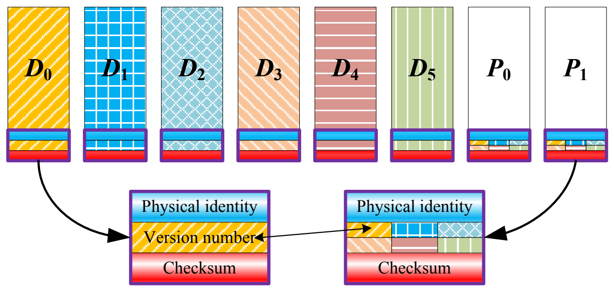
Based on the integrity primitives in Section III and the integrity checking model in Section IV, we now construct and analyze different integrity protection schemes for RAID that can detect and locate all types of silent data corruptions described in Section II-B. We first consider the baseline protection schemes that only protect data chunks, and then describe the extension for parity protection. Finally, we analyze the additional disk I/Os due to different integrity protection schemes.

A. Baseline Protection Schemes

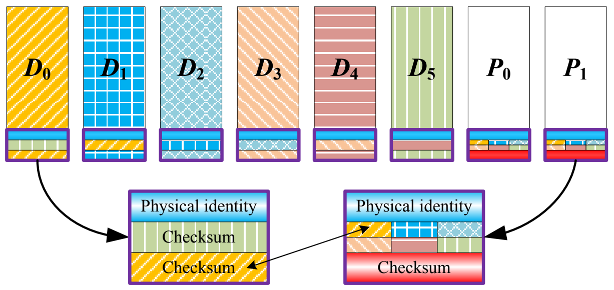
Based on Table I, we explore different combinations of integrity primitives that can detect all types of silent data corruptions. Thus, we consider the following three integrity protection schemes:

- Pure: it includes checksum mirroring only;
- Hybrid-1: it includes self-checksumming, physical identity, and version mirroring; and
- Hybrid-2: it includes self-checksumming, physical identity, and checksum mirroring.

Pure can detect all types of silent data corruptions, but it incurs high I/O overhead in general since it always uses a cross-checking operation for both the first-read and subsequent-reads. We mainly use it as a reference in this paper.



(a)



(b)

Fig. 5. Examples of (a) Hybrid-1 and (b) Hybrid-2 (with the extension for parity protection) in RAID-6.

The rationales of Hybrid-1 and Hybrid-2 are as follows. From Table I, we see that a combination of self-checksumming and physical identity can sufficiently detect misdirected writes and misdirected reads, both of which are the silent data corruptions seen by a subsequent-read. Also, both self-checksumming and physical identity are self-checking operations and do not incur any additional I/O. Thus, we construct Hybrid-1 and Hybrid-2, both of which include both self-checksumming and physical identity, and add version mirroring (for Hybrid-1) or checksum mirroring (for Hybrid-2) to cover all types of silent data corruptions. Note that Hybrid-1 is a variant of the scheme proposed in [19] by removing the logical identity that does not belong to the RAID layer, while Hybrid-2 is a new protection scheme proposed in this paper. In both Hybrid-1 and Hybrid-2, only the first-read requires a costly cross-checking operation, while each subsequent-read can use a simpler self-checking operation without any additional I/O.

Figure 5 shows the implementations of Hybrid-1 and Hybrid-2. In Hybrid-1, the checksum co-located with each data chunk should also protect the co-located physical identity and version number. On the other hand, in Hybrid-2, two checksum-based integrity primitives are employed at the same time. To reduce computational overhead, for each data chunk, we can use the same checksum for both primitives. Then the checksum should also protect the co-located physical identity. In addition, each data chunk in Hybrid-2 has two appended checksums: one for itself and another one for its neighboring chunk. To detect a torn write, the checksum for the data chunk itself should be placed at the end of the appendix.

Each of the above schemes stores the integrity metadata as an *appendix*, which is associated with each data/parity chunk. In common disk drives with standard 512-byte sector size, the appendix of each data/parity chunk is typically stored in an additional sector attached to the end of the chunk. On the other hand, in special enterprise-class disk drives with "fat" sectors whose sizes are slightly larger than 512 bytes (e.g., 520 bytes or 528 bytes), the appendix can be stored in the last sector of the chunk [3].

B. Extension for Parity Protection

We thus far focus our attention on the integrity protection of data chunks. Parity protection is necessary, since silent data corruptions on parity chunks can be returned through disk reads in two cases: (1) user write in RMW mode and (2) data reconstruction.

A straightforward extension is to reuse all the integrity primitives employed for data chunks and apply them for parity chunks. However, such an extension would cause high I/O overhead. For example, consider the case of applying version mirroring (or checksum mirroring) for each parity chunk. Then in a user write, when we update a data chunk, we have to update the version numbers (or checksums) of the data chunk, all parity chunks, as well as the additional version numbers (or checksums) of parity chunks. The updates of the additional version numbers (or checksums) will incur additional disk I/Os.

To reduce additional disk I/Os, we propose a simplified extension as follows: for both HYBRID-1 and HYBRID-2, we only use self-checksumming and physical identity on parity chunks (see Figure 5). Clearly, such an extension does not incur any additional disk I/O, since both self-checksumming and physical identity are self-checking primitives.

We argue that even with this simplified extension, all types of silent data corruptions occurring on parity chunks can still be detected. The appendix of each parity chunk now contains a physical identity, a list of version numbers or checksums for all data chunks in the same stripe, and a checksum that protects the parity chunk, the co-located physical identity, and the co-located list of version numbers (for HYBRID-1) or checksums (for HYBRID-2) (see Figure 5). The checksum of the parity chunk and the physical identity together can detect misdirected writes, misdirected reads, and torn writes. In addition, the list of version numbers or checksums for all data chunks can be used to detect lost writes to parity chunks. For example, consider a user write in RMW mode. In this case, after reading both old data chunks touched by the user write and all old parity chunks, we first self-check each old data/parity chunk using its co-located physical identity and checksum to determine if there is any corrupted chunk due to a misdirected write/read or a torn write. If no corrupted chunk is detected, then we compare the list of version numbers or checksums attached to each old parity chunk with those co-located with old data chunks to cross-check if there is any stale data or parity chunk due to a lost write. We can apply similar integrity checking for data reconstruction that also reads parity chunks.

TABLE II
NUMBER OF ADDITIONAL DISK I/OS OF DIFFERENT INTEGRITY PROTECTION SCHEMES IN DIFFERENT USER READ/WRITE SCENARIOS.

User Read/Write Scenarios			**PURE**	**HYBRID-1**	**HYBRID-2**
Partial-stripe read		*First-read*	1	1	1
		Subsequent-read	1	0	0
Partial-stripe write	*RMW mode*		1	0	2
	RCW mode	*First-read*	2	$\tau + 1$	2
		Subsequent-read	2	τ	1
Full-stripe read		*First-read*	0	1	0
		Subsequent-read	0	0	0
Full-stripe write			0	k	0

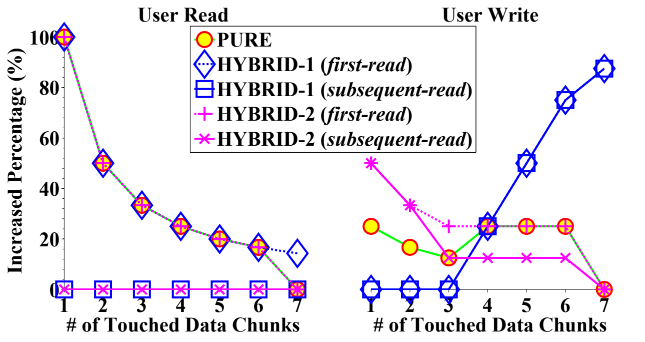
C. Analysis of Additional Disk I/Os

Table II summarizes the additional disk I/Os for our considered integrity protection schemes under different user read/write scenarios. We elaborate the details as follows.

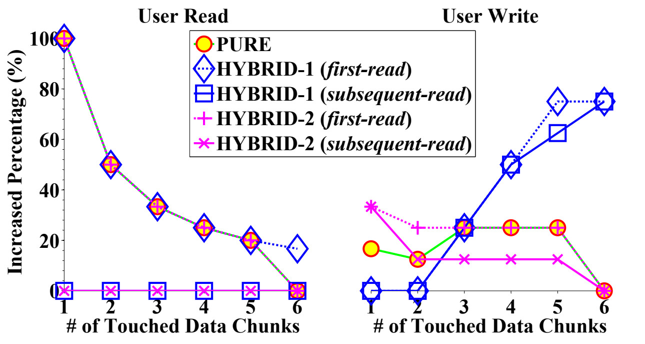
Partial-stripe user read: PURE always needs an additional disk read to the list of checksums attached to one of the parity chunks for cross-checking the data chunks touched by a user read. On the other hand, for both HYBRID-1 and HYBRID-2, the first-read needs an additional disk I/O for cross-checking, while the subsequent-read does not need any additional disk I/O for self-checking (see Section V-A).

Partial-stripe user write in RMW mode: In all protection schemes, the integrity checking can free-ride the disk reads originally triggered for reading parity chunks, and hence does not incur any additional disk I/O. However, the updates of integrity metadata may incur additional disk I/Os. PURE needs an additional disk write to the appendix of the buddy of the last data chunk touched by the user write (see Section III-D). For HYBRID-1, the updates of version numbers can free-ride the disk reads and writes originally triggered in RMW mode, so no additional disk I/O is needed. For HYBRID-2, a read-modify-write process, which involves one disk read and one disk write, is needed for updating the checksum of the last touched data chunk in the appendix of the untouched neighboring data chunk.

Partial-stripe user write in RCW mode: In this user write, the integrity checking incurs zero or one additional disk read, as in a partial-stripe user read (see discussion above). Also, the updates of integrity metadata involve additional disk I/Os, as elaborated below. PURE needs one additional disk write for updating the checksums as in RMW mode. HYBRID-1 needs τ additional disk I/Os for reading and incrementing the old version numbers co-located with the τ data chunks touched by the user write (see Section III-C). On the other hand, for HYBRID-2, only one additional disk write is needed for updating the checksum stored in the buddy of the last touched data chunk, since the read to the checksum can free-ride the original disk read to the untouched buddy data chunk in RCW mode.



(a) RAID-5 with $n = 8$ disks.



(b) RAID-6 with $n = 8$ disks.

Fig. 6. Increased percentages of disk I/Os of different protection schemes versus the number of data chunks touched by a single user read/write.

Full-stripe user read/write: First, we consider a full-stripe user read. PURE does not need any additional disk read for integrity checking, since the user read touches all data chunks and their checksums attached to buddies. HYBRID-1 still needs an additional disk I/O for cross-checking in the first-read, as in a partial-stripe user read. HYBRID-2 no longer needs any additional disk read for integrity checking in the first-read, for the same reason as PURE. We further consider a full-stripe user write, which in essence incurs no disk read. Both PURE and HYBRID-2 do not need any additional disk I/O, as all data chunks are touched by the user write, while HYBRID-1 needs k additional disk reads to the old version numbers for updates.

Analysis: We now analyze the disk I/O overhead of different integrity protection schemes for user reads/writes of different sizes. Specifically, we consider the increased percentage of disk I/Os (denoted by Ω) for a user read/write that touches τ data chunks in a stripe, where $1 \le \tau \le k = n - m$. A lower increased percentage means lower disk I/O overhead. Let δ be the number of additional disk I/Os (see Table II). For a user read, the increased percentage of disk I/Os can be directly calculated by:

$$\Omega = \frac{\delta}{\tau} \times 100\%. \tag{2}$$

For a user write, it can be either full-stripe or partial-stripe. For a full-stripe user write, the increased percentage of disk I/Os can be calculated by:

$$\Omega = \frac{\delta}{n} \times 100\%. \tag{3}$$

However, for a partial-stripe user write, it chooses either RMW or RCW modes, depending on which mode has fewer disk I/Os. Thus, the increased percentage of disk I/Os is calculated by:

$$\Omega = \left[\frac{\min(\mathcal{X}_{RMW} + \delta_{RMW}, \mathcal{X}_{RCW} + \delta_{RCW})}{\min(\mathcal{X}_{RMW}, \mathcal{X}_{RCW})} - 1 \right] \times 100\%, \tag{4}$$

where $\mathcal{X}_{RMW}$ and $\mathcal{X}_{RCW}$ are the numbers of disk I/Os for regular user writes in RMW and RCW modes, respectively (see Equation (1)), and δ_{RMW} and δ_{RCW} are the numbers of additional disk I/Os due to integrity protection in RMW and RCW modes, respectively (see Table II). We consider RAID-5 and RAID-6, both of which are configured with $n = 8$ disks. Figure 6 presents the corresponding increased percentages of disk I/Os for different integrity protection schemes. We make the following observations:

- In the user read case, both HYBRID-1 and HYBRID-2 outperform PURE mainly in a subsequent-read, since they do not introduce any additional disk I/O. Also, when the read size is smaller, the gains of HYBRID-1 and HYBRID-2 are more prominent, as the increased percentage of PURE can reach as high as 100%.
- HYBRID-1 and HYBRID-2 provide complementary I/O advantages in a user write for different sizes. Specifically, as the user write size increases, the increased percentage of disk I/Os for HYBRID-1 increases from zero to 87.5% and 75%, while that for HYBRID-2 decreases from 50% and 33.3% to zero for RAID-5 and RAID-6, respectively. In addition, the winner of HYBRID-1 and HYBRID-2 always has no more disk I/O overhead than PURE.

VI. CHOOSING THE RIGHT SCHEME

To achieve the best I/O performance, it is important to choose the most I/O-efficient integrity protection scheme for a given storage workload. Since the winner of HYBRID-1 and HYBRID-2 outperforms PURE in almost all cases, we focus on choosing between HYBRID-1 and HYBRID-2. In this section, we propose a quantitative method that effectively chooses between HYBRID-1 and HYBRID-2 for a given storage workload. We also discuss how to implement the quantitative method in real deployment.

A. *Quantitative Method*

From the results of Figure 6, the difference in I/O overhead between HYBRID-1 and HYBRID-2 mainly lies in user writes. Specifically, let τ be the number of touched data chunks in a user write. As τ increases, the I/O overhead of HYBRID-1 increases from zero to a large value, while that of HYBRID-2 is opposite and decreases from a large value to zero. We can thus define a switch point τ^* as the threshold of the user write size to switch between HYBRID-1 and HYBRID-2, such that if $\tau \le \tau^*$, we choose HYBRID-1; if $\tau > \tau^*$, we choose HYBRID-2.

In addition, near the switch point τ^*, the user write is a partial-stripe one. From the results on partial-stripe user writes in Table II (see Section V-C), HYBRID-1 has fewer additional disk I/Os in RMW mode, while HYBRID-2 has fewer additional disk I/Os in RCW mode. Thus, the switch between HYBRID-1 and HYBRID-2 also leads to a switch between RMW and RCW modes. Specifically, if $\tau \leq \tau^*$, HYBRID-1 will be selected and it will operate in RMW mode; if $\tau > \tau^*$, HYBRID-2 will be selected and it will operate in RCW mode.

We now derive τ^*. Note that the number of disk I/Os of each of HYBRID-1 and HYBRID-2 is the sum of the disk I/Os in a regular user write (see Equation (1)) and the additional disk I/Os due to integrity protection (see Table II). If $\tau \leq \tau^*$, HYBRID-1 is used and it operates in RMW mode, so the number of disk I/Os (denoted by $\mathcal{X}_L$) is:

$$\mathcal{X}_L = 2(\tau + m). \tag{5}$$

On the other hand, if $\tau > \tau^*$, HYBRID-2 is used and it operates in RCW mode, so the number of disk I/Os (denoted by $\mathcal{X}_R$) is:

$$\mathcal{X}_R = n + 1 + \sigma, \tag{6}$$

where σ is 1 for the first-read case, or 0 for the subsequent-read case. By setting $\mathcal{X}_L = \mathcal{X}_R$, we obtain τ^*:

$$\tau^* = \left\lceil \frac{n+1}{2} \right\rceil - m. \tag{7}$$

We next show how to quantitatively choose between HYBRID-1 and HYBRID-2 for a storage workload. Consider a RAID array configured with n disks, m parity chunks per stripe, and r sectors per chunk. Given that the standard sector size is 512 bytes, the chunk size, denoted by S_{chunk}, is $S_{chunk} = 0.5 \times r$ KB. In addition, suppose the average write size of a storage workload is $\overline{S}_{write}$, which can be estimated by tracing the workload patterns [11, 16, 30, 34]. This implies that the average number of data chunks touched by a user write is $\overline{S}_{write}/S_{chunk}$. Thus, our quantitative method chooses between HYBRID-1 and HYBRID-2 according to the following rule: If

$$\frac{\overline{S}_{write}}{S_{chunk}} \leq \left\lceil \frac{n+1}{2} \right\rceil - m, \tag{8}$$

we choose HYBRID-1; while if

$$\frac{\overline{S}_{write}}{S_{chunk}} > \left\lceil \frac{n+1}{2} \right\rceil - m, \tag{9}$$

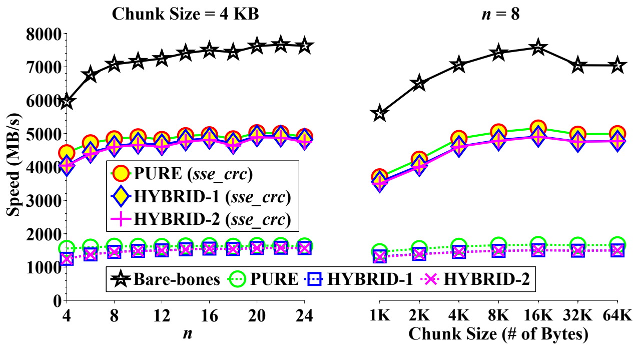
we choose HYBRID-2.

Based on the above rule, we deduce the following two change trends:

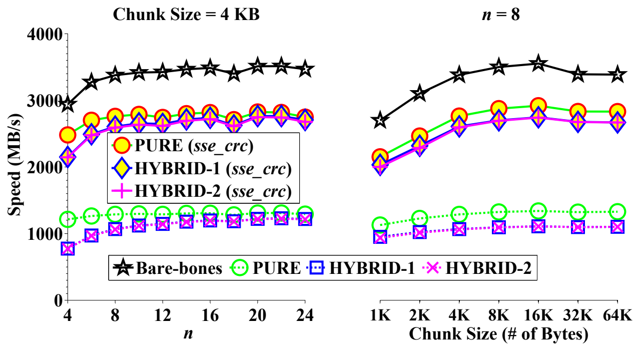
- With the increase of either $\overline{S}_{write}$ or m, we switch from HYBRID-1 to HYBRID-2;
- With the increase of either S_{chunk} or n, we switch from HYBRID-2 to HYBRID-1.

B. Discussion

In practice, we can adopt our proposed quantitative method based on workload characterization [11, 16, 30, 34]. We first analyze the workload traces collected beforehand to obtain the average write size, and then use the average write size to determine the most I/O-efficient scheme (i.e., either HYBRID-1 or HYBRID-2). The selected scheme is finally configured in the RAID array during initialization. Our assumption here is that while the write size varies across applications, the variability of the write size within each application is low [30]. Thus, the average write size can be accurately predicted through workload characterization.



(a) RAID-5



(b) RAID-6

Fig. 7. Encoding speeds of RAID integrated with different integrity protection schemes. The remark "*sse_crc*" indicates the implementation with the SSE4.2 `CRC32` instruction.

VII. EVALUATION

We evaluate and compare the integrity protection schemes, and address the following issues. First, we examine the computational overhead due to integrity metadata computations. Second, we examine the disk I/O overhead under various real-world storage workloads that have different read/write characteristics. Finally, we evaluate the effectiveness of our quantitative method for choosing the right integrity protection scheme (see Section VI).

A. Computational Overhead

We note that the integrity protection schemes incur computational overhead in metadata computations (e.g., checksum computations) in general RAID operations. Here, we focus on RAID encoding, which needs to compute all integrity metadata in each stripe.

We implement RAID encoding coupled with different integrity protection schemes. We use the libraries GF-Complete [27] and Crcutil [15] to accelerate the calculations of parity

TABLE III

INFORMATION OF THE STORAGE WORKLOAD TRACES.

I/O Traces	Trace name	DAP-DS	TPC-E	LM-TBE	MSN-BEFS	MSN-CFS	DTRS	TPC-C	DAP-PS	Exch	WBS	RAD-BE	RAD-AS
	Device ID	0	13	1	5	2	6	2	0	4	0	4	0
	Duration (Hrs)	24	0:17	24	6	6	24	0:06	24	24	24	18	18
User Reads	Proportion (%)	90.49	90.40	79.46	76.58	73.96	65.42	62.44	56.19	52.58	50.71	18.49	10.43
	Avg. size (KB)	31.50	8.00	53.28	9.88	7.55	21.60	8.00	62.13	10.04	26.09	152.28	10.29
User Writes	Proportion (%)	9.51	9.60	20.54	23.42	26.04	34.58	37.56	43.81	47.42	49.29	81.51	89.57
	Avg. size (KB)	7.04	9.26	63.47	10.45	13.63	34.57	8.62	97.13	12.59	29.78	8.54	8.24

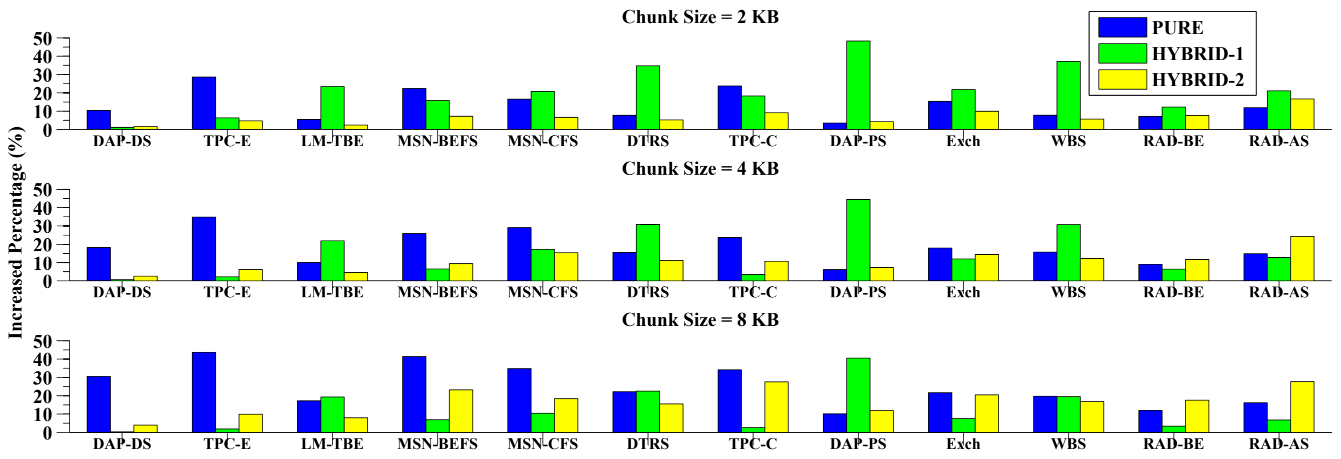


Fig. 8. Increased percentages of disk I/Os of different integrity protection schemes under different storage workloads, using RAID-6 with $n = 8$ disks.

chunks and checksums, respectively. For checksum computations, we employ the CRC-32C (Castagnoli) polynomial [18] that has been included in the SSE4.2 `CRC32` instruction [14]. We also implement bare-bones RAID, and the integrity protection schemes without using SSE4.2 `CRC32` for reference. We run all the tests on an Intel Xeon E5530 CPU running at 2.40GHz with SSE4.2 support.

Figure 7 presents the encoding speed results for RAID-5 and RAID-6 for different values of n (with chunk size 4KB) and different chunk sizes (with $n = 8$). Although integrity protection with SSE4.2 `CRC32` support still has lower encoding speed than bare-bones RAID (e.g., by up to 37.76% for RAID-5 and up to 27.05% for RAID-6), the encoding speed can still reach over 4GB/s for RAID-5 and 2.5GB/s for RAID-6, both of which are much faster than the access speed of modern disk drives (typically below 1GB/s). Note that SSE4.2 `CRC32` significantly increases the encoding speed of integrity protection (by two to three times). Thus, even with integrity protection, the RAID performance is still bottlenecked by disk I/Os, rather than the CPU.

B. I/O Overhead for Storage Workloads

We evaluate the disk I/O overhead of different integrity protection schemes for various storage workloads with different I/O characteristics.

Evaluation methodology: We develop a simulator that can measure the increased percentage of disk I/Os of each integrity protection scheme, with the inputs of RAID configuration parameters (i.e., n, m, and r) and storage workload traces. The simulator counts the numbers of disk I/Os for PURE, HYBRID-1, and HYBRID-2, and that for bare-bones RAID, by emulating the mapping from user reads/writes to disk I/Os in real RAID arrays. For HYBRID-1 and HYBRID-2, the simulator maintains a bitmap to record the read status of each data chunk in a RAID array, so as to differentiate between the first-read and subsequent-reads (see Section IV-B).

Our trace-driven simulation uses 12 sets of storage workload traces collected from the production Windows servers [16]. Since each set of traces is collected from several storage devices (each device is either a disk drive or a disk array), we pick the traces from only one busy device. Table III summarizes the user read/write characteristics of different traces, sorted by their read proportions in descending order. We see in most storage workloads, the read proportion is higher than 50%.

Evaluation results: We evaluate the disk I/O overhead of different integrity protection schemes over bare-bones RAID under different storage workloads. In the interest of space, we only present our findings for RAID-6 with $n = 8$ disks, while similar observations are made for RAID-5 and other values of n. Figure 8 shows three groups of simulation results with chunk sizes 2KB, 4KB, and 8KB. We make the following observations:

- Owing to the complementary performance gains of HYBRID-1 and HYBRID-2 for different kinds of storage workloads, the I/O overhead can be kept at a reasonably low level (often below 15%), as long as the right scheme between HYBRID-1 and HYBRID-2 is chosen for each kind of storage workloads.

- The winner of HYBRID-1 and HYBRID-2 often incurs lower I/O overhead than PURE, because of the I/O gain in subsequent-reads (see Section V-C). The gain can be more prominent for storage workloads with a larger read proportion and a smaller average read size. For example, for TPC-E with a read proportion 90.40% and an average read size 8.00KB, the increased percentage of disk I/Os of the winner between HYBRID-1 and HYBRID-2 is only 1.86% in the case of 8KB chunk size, while that of PURE can reach up to 43.74%.
- We observe some glaringly high I/O overhead in PURE and HYBRID-1: PURE can incur very high I/O overhead (up to 43.74%) in read-heavy workloads, especially if the workloads mostly consist of small reads (e.g., TPC-E); HYBRID-1 can incur very high I/O overhead (up to 48.28%) in the workloads with large writes (e.g., DAP-PS). On the other hand, HYBRID-2 incurs relatively moderate I/O overhead (no more than 27.70%).
- With a small chunk size (e.g., 2KB), HYBRID-2 has the minimum increased percentage of disk I/Os in most cases. As the chunk size increases, the I/O overhead of HYBRID-2 also increases, while the I/O overhead of HYBRID-1 decreases and may become significantly lower than that of HYBRID-2, especially for storage workloads with a smaller average write size. Thus, the pre-configuration of chunk size of a RAID array can significantly influence the choice between HYBRID-1 and HYBRID-2 (see Section VI).

C. *Effectiveness of Choosing the Right Scheme*

We evaluate the effectiveness of our proposed quantitative method (see Section VI) for choosing between HYBRID-1 and HYBRID-2, based on the simulation results of I/O overhead presented in the last subsection.

For each of the 36 cases in Figure 8, we compare if the choice made by our quantitative method according to the information of average write size in Table III is consistent with the choice made directly from the simulation results. We find that 34 out of 36 comparison outcomes (or 94.44%) are consistent. The quantitative method only makes two inconsistent choices for DAP-DS with 2KB chunk size and Exch with 4KB chunk size. From Figure 8, we see that for each of the two inconsistent cases, the I/O overhead difference between HYBRID-1 and HYBRID-2 is insignificant (below 3%). Thus, our quantitative method can be very effective in choosing between HYBRID-1 and HYBRID-2 to achieve I/O-efficient integrity protection in a real-world scenario.

VIII. RELATED WORK

Silent data corruptions in disk drives have been widely recognized in the literature. Two field studies from CERN [17] and NetApp [3] reveal the significant occurrences of silent data corruptions in production storage systems.

In Section III, we provide a taxonomy of integrity primitives that employ special integrity metadata. Several studies analyze the metadata-based integrity primitives from different perspectives. Sivathanu *et al.* [38] survey three common integrity techniques including mirroring, RAID parity checking, and checksumming, and then provide a general discussion of their applications and implementations in all layers of storage systems. Hafner *et al.* [10] discuss the causes and manifestations of silent data corruptions in disk drives, and propose a family of parity appendix methods that arrange different general layouts of integrity metadata in RAID arrays. Krioukov *et al.* [19] propose model checking to analyze the effectiveness of different data protection strategies in RAID arrays. Rozier *et al.* [33] develop a simulation model to evaluate silent data corruptions in petabyte-scale storage systems. Unlike previous work, we focus on the performance perspective, from which we quantitatively analyze how to combine different metadata-based integrity primitives to reduce I/O overhead.

There are other integrity primitives that do not employ any integrity metadata but can be deployed in RAID. *Write-verify* [31, 40] rereads the data that has just been written to disk and checks the correctness of the written data. However, it causes an additional disk read in each disk write and also needs a certain amount of cache space for temporarily storing the written data. Also, it cannot detect misdirected writes due to the mismatch of the reread and misdirected write positions. *Parity checking* [1, 20, 21] recomputes a parity chunk from all data chunks in the same stripe and compares the regenerated parity chunk with that stored on disk. However, checking a single data chunk involves all other data chunks in the same stripe, and thus incurs very high I/O overhead. In this paper, we do not consider write-verify and parity checking due to their high I/O overhead.

In addition to the work in the RAID layer, some studies address silent data corruptions in the upper layers to achieve end-to-end data integrity. For example, some studies cope with silent data corruptions in file systems, such as IRON [29], ZFS [4], and BTRFS [32]. However, file systems are vulnerable to memory corruptions [13, 37, 39], which require additional data protection above the file system layer [42, 43]. Other studies [8, 9, 12, 24] develop end-to-end data integrity solutions based on the integration of the T10 Protection Information (PI) industry standard [40] and the Data Integrity Extension (DIX) jointly developed by Oracle and Emulex [25]. T10 PI defines the data integrity protection for enterprise-class SCSI-based disk drives, covering the data path from the host bus adapter (HBA) through the RAID controller to the disk drives, while DIX extends the integrity protection to cover the data path from the application to the HBA. A variant of T10 PI for SATA-based drives is also implemented [22]. In fact, our work can co-exist with both T10 PI and DIX approaches, as we target silent data corruptions in the disk drives, while the latter cope with data corruptions on the data transmission path between the application and the disk drives. How to combine the T10 PI and DIX approaches and our work is posed as future work.

IX. Conclusions

Integrity protection is critical for protecting RAID against silent data corruptions, but its I/O overhead can be an obstacle to its practical deployment. We conduct a systematic study on how to protect RAID against silent data corruptions with low I/O overhead. We show via an integrity checking model that we can employ simpler and more I/O-efficient integrity checking mechanisms for a large proportion of disk reads. This idea enables us to design I/O-efficient integrity protection schemes, while still being capable of detecting all types of silent data corruptions. We propose two integrity protection schemes that provide complementary performance advantages, and further develop a quantitative method for effectively choosing between the two schemes. Our trace-driven simulation results show that with the appropriate integrity protection scheme, we can reduce the I/O overhead to below 15%.

Acknowledgements

This work was supported in part by grants AoE/E-02/08 and ECS CUHK419212 from the UGC/RGC of Hong Kong.

References

[1] H. P. Anvin. The mathematics of RAID-6, Dec. 2011. https://www.kernel.org/pub/linux/kernel/people/hpa/raid6.pdf.

[2] L. N. Bairavasundaram, G. R. Goodson, S. Pasupathy, and J. Schindler. An analysis of latent sector errors in disk drives. In *Proc. of ACM SIGMETRICS*, 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 *Proc. of USENIX FAST*, Feb. 2008.

[4] J. Bonwick. ZFS end-to-end data integrity, Dec. 2005. https://blogs.oracle.com/bonwick/entry/zfs_end_to_end_data.

[5] 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, 1994.

[6] B. E. Clark, F. D. Lawlor, W. E. Schmidt-Stumpf, T. J. Stewart, and J. George D. Timms. Parity spreading to enhance storage access, Aug. 1988. U.S. Patent 4761785.

[7] M. H. Darden. Data integrity: The Dell|EMC distinction, May 2002. http://www.dell.com/content/topics/global.aspx/power/en/ps2q02_darden?c=us.

[8] EMC Corporation. An integrated end-to-end data integrity solution to protect against silent data corruption. White Paper H10058, Oct. 2012. https://community.emc.com/servlet/JiveServlet/downloadBody/19278-102-2-68831/h10058-data-integrity-solution-wp.pdf.

[9] Fusion-io, Inc. Oracle OpenWorld 2013 Demo: T10 PI data integrity demonstration protecting against silent data corruption, Nov. 2013. http://www.fusionio.com/t10-pi-data-integrity-demonstration-protecting-against-silent-data-corruption/.

[10] J. L. Hafner, V. Deenadhayalan, W. Belluomini, and K. Rao. Undetected disk errors in RAID arrays. *IBM Journal of Research and Development*, 52(4/5):413–425, 2008.

[11] W. W. Hsu and A. J. Smith. Characteristics of I/O traffic in personal computer and server workloads. *IBM Systems Journal*, 42(2):347–372, 2003.

[12] Huawei Technologies Co., Ltd. End-to-end data integrity protection in storage systems. Technical Whitepaper, Aug. 2013. http://www.huawei.com/ilink/cnenterprise/download/HW_309360.

[13] A. A. Hwang, I. Stefanovici, and B. Schroeder. Cosmic rays don't strike twice: Understanding the nature of DRAM errors and the implications for system design. In *Proc. of ACM ASPLOS*, Mar. 2012.

[14] Intel Corporation. Intel$^{\circledR}$ SSE4 programming reference. Reference Number: D91561-001, Apr. 2007.

[15] A. Kadatch and B. Jenkins. Crcutil — High performance CRC implementation, Jan. 2011. http://code.google.com/p/crcutil/.

[16] S. Kavalanekar, B. Worthington, Q. Zhang, and V. Sharda. Characterization of storage workload traces from production Windows servers. In *Proc. of IEEE Int. Symp. on Workload Characterization (IISWC)*, Sept. 2008.

[17] P. Kelemen. Silent corruptions. Presentation Slides, presented at the 8th Annual Workshop on Linux Cluster for Super Computing (LCSC '07), Oct. 2007. http://www.nsc.liu.se/lcsc2007/presentations/LCSC_2007-kelemen.pdf.

[18] P. Koopman. 32-bit cyclic redundancy codes for internet applications. In *Proc. of IEEE/IFIP DSN*, June 2002.

[19] A. Krioukov, L. N. Bairavasundaram, G. R. Goodson, K. Srinivasan, R. Thelen, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Parity lost and parity regained. In *Proc. of USENIX FAST*, Feb. 2008.

[20] M. Li and J. Shu. Preventing silent data corruptions from propagating during data reconstruction. *IEEE Transactions on Computers*, 59(12):1611–1624, Dec. 2010.

[21] J. Luo, C. Huang, and L. Xu. Decoding STAR Code for tolerating simultaneous disk failure and silent errors. In *Proc. of IEEE/IFIP DSN*, June 2010.

[22] NEC Corporation. Silent data corruption in disk arrays: A solution. White Paper WP116-2_0909, Sept. 2009. http://www.necam.com/docs/?id=54157ff5-5de8-4966-a99d-341cf2cb27d3.

[23] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proc. of ACM SIGMOD*, Jun 1988.

[24] M. Petersen and S. Singh. How to prevent silent data corruption — best practices from Emulex and Oracle, Feb. 2013. http://www.oracle.com/technetwork/articles/servers-storage-dev/silent-data-corruption-1911480.html.

[25] M. K. Petersen. Linux data integrity extensions. In *Proc. of Ottawa Linux Symposium (OLS)*, July 2008.

[26] E. Pinheiro, W.-D. Weber, and L. A. Barroso. Failure trends in a large disk drive population. In *Proc. of USENIX FAST*, Feb. 2007.

[27] J. S. Plank, K. M. Greenan, and E. L. Miller. Screaming fast Galois Field arithmetic using Intel SIMD instructions. In *Proc. of USENIX FAST*, Feb. 2013.

[28] J. S. Plank and C. Huang. Tutorial: Erasure coding for storage applications. Slides presented at FAST-2013, Feb. 2013.

[29] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. IRON file systems. In *Proc. of ACM SOSP*, Oct. 2005.

[30] A. Riska and E. Riedel. Disk drive level workload characterization. In *Proc. of USENIX ATC*, May 2006.

[31] A. Riska and E. Riedel. Idle read after write - IRAW. In *Proc. of USENIX ATC*, June 2008.

[32] O. Rodeh, J. Bacik, and C. Mason. BTRFS: The Linux B-Tree filesystem. *ACM Transactions on Storage*, 9(3):1–32, 2013.

[33] E. W. D. Rozier, W. Belluomini, V. Deenadhayalan, J. Hafner, K. Rao, and P. Zhou. Evaluating the impact of undetected disk errors in RAID systems. In *Proc. of IEEE/IFIP DSN*, June 2009.

[34] C. Ruemmler and J. Wilkes. Unix disk access patterns. In *Proc. of Usenix Winter*, Jan. 1993.

[35] B. Schroeder, S. Damouras, and P. Gill. Understanding latent sector errors and how to protect against them. In *Proc. of USENIX FAST*, Feb. 2010.

[36] B. Schroeder and G. A. Gibson. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you? In *Proc. of USENIX FAST*, Feb. 2007.

[37] B. Schroeder, E. Pinheiro, and W.-D. Weber. DRAM errors in the wild: A large-scale field study. In *Proc. of ACM SIGMETRICS/Performance*, June 2009.

[38] G. Sivathanu, C. P. Wright, and E. Zadok. Ensuring data integrity in storage: Techniques and applications. In *Proc. of ACM StorageSS*, Nov. 2005.

[39] V. Sridharan and D. Liberty. A study of DRAM failures in the field. In *Proc. of IEEE/ACM SC*, Nov. 2012.

[40] T10 Technical Committee. SCSI block commands - 3 (SBC-3). Working Draft T10/BSR INCITS 514 Revision 35k, Oct. 2013.

[41] A. Thomasian and M. Blaum. Higher reliability redundant disk arrays: Organization, operation, and coding. *ACM Transactions on Storage*, 5(3):1–59, 2009.

[42] Y. Zhang, D. S. Myers, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Zettabyte reliability with flexible end-to-end data integrity. In *Proc. of IEEE MSST*, May 2013.

[43] Y. Zhang, A. Rajimwale, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. End-to-end data integrity for file systems: A ZFS case study. In *Proc. of USENIX FAST*, Feb. 2010.