Verification of Parity Data in Large Scale Storage Systems

Thomas Schwarz, S.J.
Department of Computer Engineering
Santa Clara University
Santa Clara, CA 95053, USA
tjschwarz@scu.edu

Abstract

Highly available storage uses replication and other redundant storage to recover from a component failure. If parity data calculated from an erasure correcting code is not updated or becomes otherwise corrupted, recovery from a failure does not recover the correct data but mostly garbled data. This paper presents an algebraic signature scheme that can detect parity discrepancies for parity calculated with XORing, generalized Reed-Solomon codes, or convolutional array codes. Maintaining and checking the signature of client and parity data allows us to ensure coherence in the storage system and thus to accurately rebuild data on lost devices. Our scheme is combined with disk scrubbing, necessary to detect masked disk failures.
Keywords: Highly available storage system. Redundancy group coherence. Reed-Solomon codes. Convolutional array codes.

1. Introduction

As disks outpace tapes in capacity increases, large scale storage systems based on disks have reached the planning stages at institutions such as the U.S. National Laboratories. Such a system encompasses several thousand disks and reaches a storage capacity of 1-5 Petabytes (PB). Despite its large size and despite its use of commodity disks, the availability of files needs to be very high, since data often cannot be reproduced or only at large costs. For example, the system might store terabytes of data striped over many disks from a single simulation.

The large number of devices in such a system increases the likelihood of failures. For instance, error rates of one undetected, repeated read error of 1 in 10^{15} bits on a commodity disk are common. A single disk running at 25MB/sec would experience such an error about once a year. In a large scale storage system with 10,000 disks, such an error will occur once per hour somewhere in the system. Many of the error modes are masked such as this one, that is, we only detect them when we try to read a block of data, or even more insidiously and fortunately much more rarely, if we try to use the data in the block.

To protect against the effect of failures, we store data redundantly. For example, we store two copies of each datum (mirroring), or we group data blocks together in a *redundancy group* to which we add a parity block that contains the bitwise xor (the parity) of the data blocks. The latter is the scheme used in RAID Level 5. To achieve higher levels of availability, we can place the same data block into a number of redundancy groups, or even simpler, add more than one parity block to the redundancy group. The latter involves using an erasure correcting code (ECC). To clarify our argument, we also call a group of mirrored or replicated blocks a redundancy group.

The large number of devices in the system not only increases the likelihood of failures but forces us to pay attention to failure modes that are negligible for traditional disk arrays with less than one hundred disks. This paper focuses on one of these concerns, namely how to detect if incoherency within a redundancy group. If the coherence of redundant data is lost, then data rebuild after a failure does not recover the data lost but some seemingly random value.

To illustrate the redundant data coherency problem, assume that we mirror data. A device with one of the three replicas fails. Because of an undetected, repeatable read error, the two remaining replicas differ. To decide the true copy, we need to investigate their meaning, that is, by finding out which of the two values makes sense. This is sometimes impossible, and most often requires human intervention. The problem becomes worse if we use ECC.

We solve the problem of undetected client and parity data corruption with a scheme based on signatures with

algebraic properties. Signatures or hashes are small strings calculated from the contents of a storage block. They change when the block is slightly altered and two arbitrary blocks have the same signature only with probability 2^{-f}, where f is the length of the signature in bits. Our signatures allow us to calculate the signature of a parity block (such as produced by RAID 5) or a generalized parity block (produced by an erasure correcting code such as Reed-Solomon or an array code) from the signatures of the data blocks.

To protect data integrity, we maintain a signature for each data or parity block in the disk farm, stored on the disk itself. Periodically, each disk checks whether the signature faithfully reflects the block contents. Also periodically, each disk checks whether for any redundancy group (formed by client data blocks and associated parity blocks) it stores, the signatures prove that parity blocks reflect the data. Our scheme works for n-way mirroring, RAID Level 5 like parity groups with a single (XOR-)parity block, and RAID Level 6 like parity groups with more than one parity block that use generalized Reed Solomon codes or convolutional array codes.

2. Disk Based Large Storage Systems

Disk arrays have gained increasing market share and are expanding into near-line systems that were once served with tape drives. The internet archive [9] is an example of such a facility that stores over 100 TB of compressed data on approximately 150 desktop computers with four hard drives each. In order to deal with the likelihood of device failure in systems of this and larger size, we introduce some type of redundancy into the storage of data. A prevalent choice is mirroring as implemented in RAID Level 1, 10, 0/1 etc. Calculating and storing parity is another important technique, in which the bitwise parity (the exclusive or) of a number of data blocks is stored on yet another disk. Since any data block is also the parity of all other data blocks and this parity block, it is possible to reconstruct any lost data block as long as all the others are still available. Raid Levels 4 and 5 use this scheme. For systems in the PB range, we need to use a higher level of protection than provided by simple RAID [3], [18], [19]. To do so, we can group blocks on different disks in a redundancy group like before, but add two or more parity blocks to each redundancy group. We then use *Erasure Correcting Codes (ECC)* to calculate these parity blocks. We describe two good ECC in Section 5. RAID Level 6 uses this scheme. Alternatively, we can place blocks in more than one redundancy group and add a parity block for each of the redundancy groups. In general, we place m blocks of user data into a *redundancy group*, add k *parity blocks* to the group, so that all $n = m + k$ blocks reside on different disks and so that access to any m blocks (whether parity or data) allows us to rebuild any lost block. We refer to such a scheme as an m/n scheme and call it k available.

3. Problem Definition

3.1. Masked Disk Drive Failures and the Data Parity Coherency Problem

Data in a storage system faces three threats: outright disk failure, disk block failure (when we cannot read a block on an otherwise functioning disk even with multiple trials), and data corruption. Device failures are easy to detect and we deal with device failure by using the storage redundancy to rebuild the contents of the drive on one or more other drives. We detect block failures by performing *disk scrubbing*, that is, reading periodically all the data on the disk. Once a block defect has been detected, we can rebuild the data on the affected block on another disk or even in the same spot. (Sometimes an off-track write destroys data on an adjacent track, but the affected block of course be reused.)

Block failures can arise in a number of ways. For example, the magnetic media might deteriorate in a spot, maybe because a particle was wedged between the head and the media, scratched the surface, and permanently destroyed the capability of the media there to retain magnetic data [7]. Occasionally, a disk suffers bit rot, that is, one or a few bits flip. For example, at current areal densities achieved on production hard drives, traces of the super-paramagnetic effect (thermodynamic instability of small magnetic spots) can already be detected on every drive. In the vast majority of cases, an internal checksum flags an affected block as unreadable, when accessed. Sometimes, this bit rot is not detected and corrupted data is given to the user. A much more potent source of data corruption is of course software failure. For example, a write operation might not be performed at all the disks that need to be written and leave the application data in an incoherent state.

Media errors and data corruption are masked failures, they only become apparent when we try to use the affected data. Data corruption is more insidious, because we need to recognize the corruption. This is difficult enough for user data, but we can only discover the corruption of parity data by comparing it with the user data that it protects. This is the parity-data coherency problem. It can prevent us from successfully

rebuilding otherwise lost data. Consider the following two scenarios:

***Scenario 1**:* We store three replicas of datum A: A_0, A_1, and A_2. Assume replica A_1 is lost and that replica A_0 and A_2 differ. Which one is the true copy which we want to replicate twice? We need to base our decision on whether A_0 or A_2 make sense. If both make sense, we are stuck.

Scenario 2: Assume that we use a $n/(n+2)$ scheme, that is, that we group n data blocks (or objects) D_0, D_1, … D_{n-1} into a redundancy group and add parity blocks (or objects) P_0 and P_1. Of course, all data and parity blocks are located on different disks. Now assume that D_1 has suffered data corruption, but is still readable. Assume that the device carrying D_0 fails. We can recover by assessing any n of the $n+1$ remaining blocks. But if we include D_1 into the mix, then the recovered version D_0, let us call it D_0', is *not* the original D_0. The true D_0 is still recoverable, but only by using D_2, D_3,…, P_0, P_1. Again, we can only make a decision based on whether the recovered D_0 is meaningful or by checking D_1,…D_{n-1} directly. This scenario shows that unmasking the corruption at D_1 has a direct effect on the availability of the otherwise unrelated data blocks D_0, D_2,… D_{n-1}.

The redundancy group coherency problem can arise not only from bit rot, but also from faulty software that on rare occasions might fail to update some parity data when client data is written or vice versa. As we have seen, this affects the reliability of *all* data blocks in the redundancy group.

To summarize, in a highly available storage system, we need to discover quickly masked failures such as media failures and data corruption. In addition, we need to maintain the coherency of data within a redundancy group.

3.2 Signature Schemes

Our solution *flags* corrupted or incoherent data in the hope that quick detection allows us to repair the damage before any lasting harm is done. We associate a signature with each data and parity block. These blocks can be quite large, e.g. 1GB. The determination of the optimal size is beyond the scope of this paper, but see [18], [19]. In contrast, our signatures are quite small, as small as 4B or 8B. The signature (a.k.a. hash, checksum) is a bit string of fixed length l that is calculated from the contents of the block. We could use cryptographically strong hashes such as MD5 and SHA1 as in Tripwire [11] to verify block contents, but this would not protect us from incoherent parity data.

We use an "algebraic signature" defined below that we store with each block. The signature is *guaranteed* to change with small changes in the block. Two random blocks have the same signature only with probability 2^{-f} (where f is the length of the signature in bits). The algebraic properties of the signature allow us to calculate the signature of an updated block from the update information only. They also allow us to calculate the signature of parity blocks, whether generated by parity as in RAID 5, or by generalized Reed-Solomon or the convolutional array codes.

4. Algebraic Signature Definition and Properties

4.1. Galois Field Operations

Our algebraic signatures use Galois field calculations. The elements of this Galois field are symbols, that is, bit strings of length f. Typically, $f = 8$, and then a symbol is simply a byte. Sometimes it is advantageous to use $f = 16$ (half-words) or even $f = 32$ (words). We can add, multiply, and divide symbols just as we manipulate real numbers or rational numbers (fractions). There are two special elements, the zero element 0 and the one element 1, given by the bit strings 0000 … 0000 and 0000 … 0001. The addition is simply the bitwise XOR of the string. The multiplication is more involved and several multiplication algorithms are known. We use a method based on logarithm and antilogarithm table. It is fairly easy to find an element α in the Galois field such that all non-zero elements are powers of this α. Such elements are called *primitive*. If $\beta = \alpha^i$, $0 \le i \le 2^f-1$, then we write $i = \log_\alpha(\beta)$ and $\beta = \text{antilog}_\alpha(i)$. For nonzero γ and η we then have

$$\gamma \cdot \eta = \text{antilog}_\alpha(\log_\alpha(\gamma) + \log_\alpha(\eta)).$$

A product with one or both factors zero is of course zero. We implement now Galois field multiplication through table look-ups. The size of the tables in our implementation [13] is a moderate $3 \cdot 2^f$ for $f = 8, 16$. For $f = 32$, a multiplication can be done using five multiplications in the smaller Galois field with $f = 16$. See [13] or [14] for a more detailed explanation. While we define signatures in terms of Galois field operations (Section 4.2), we actually do not use multiplications when we calculate signatures (Section 4.3).

4.2. Signature Definition

A *block P* is a string of n symbols p_i, $1 \le i \le l$, which in turn are bit strings of length f. In our case, the symbols p_i are bytes or 2-byte words. The symbols are elements of the Galois field, GF (2^f).

Definition 1: Let α be a primitive element of the Galois field GF(2^f), $P = (p_1,p_2,...p_l)$ be a block, and m be an integer ≥ 2. Then define

$$\mathrm{sig}_\alpha(P) := \sum_{\nu=1}^{l} \alpha^{\nu-1} p_\nu$$

$$\mathrm{sig}_{\alpha,m}(P) := (\mathrm{sig}_1(P), \mathrm{sig}_\alpha(P), \mathrm{sig}_{\alpha^2}(P)..., \mathrm{sig}_{\alpha^{m-1}}(P))$$

We call $\mathrm{sig}_{\alpha,m}$ the m-signature with base α.

The m-signature is a vector with m coordinates, each of length f bits, so that the combined signature is mf bits long. The first coordinate of the m-signature is the XOR of the symbols in the block. This coordinate is included because of the ease of calculation, but not essential to the properties of the signature, which could alternatively consists of the $\mathrm{sig}_\beta(P)$ with $\beta=\alpha^1$, ... α^m. Definition 1 is related, but not identical to the one in [2] and [10]. We now list a number of properties of our algebraic signatures:

Proposition 1: If the block length l is smaller than $2^f - 1$, then the m-signature with base α discovers any changes of up to m symbols.

Proposition 2: The probability that two signatures of two random blocks coincide is 2^{-mf}.

Proposition 3: Let us change block $P = (p_1, p_2, \ldots p_l)$ to block P' where we replace the symbols starting in position r and ending with position s-1 with the string $q_r, q_{r+1}, \cdots, q_{s-1}$. We define the *Δ-string* as $\Delta = (\delta_0, \delta_1, \ldots \delta_{s-r-1})$ with $\delta_i = p_{r+i} - q_{r+i}$. Then for any $\beta \in \mathrm{GF}(2^f)$:

$$\mathrm{sig}_\beta(P') = \mathrm{sig}_\beta(P) + \beta^r \mathrm{sig}_\beta(\Delta).$$

Proposition 4: If we concatenate block P_1 of length l with block P_2, then we have

$$\mathrm{sig}_\beta(P_1 \mid P_2) = \mathrm{sig}_\beta(P_1) + \beta^l \mathrm{sig}_\beta(P_2).$$

Proposition 5: Let $P^{(i)}$, $1 \le i \le r$, be r blocks, and let $P^{(\mathrm{parity})}$ be the block generated as the parity of the blocks $P^{(i)}$, $1 \le i \le r$. That is, the symbols of $P^{(\mathrm{par})}$ are calculated as $p^{(\mathrm{par})}{}_i = p^{(1)}{}_i \oplus p^{(2)}{}_i \oplus \ldots \oplus p^{(\mathrm{r})}{}_i$. Then $\mathrm{sig}_{\alpha,n}(P^{(\mathrm{par})}) = \mathrm{sig}_{\alpha,n}(P^{(1)}) \oplus \mathrm{sig}_{\alpha,n}(P^{(2)}) \oplus ... \oplus \mathrm{sig}_{\alpha,n}(P^{(r)})$ where the xor of vectors is taken coordinate-wise.

Our signature also detects swapping two sub-blocks in the same block [14], [16]. The proofs of propositions 1-5 is in [14]. Prop. 2 says that the probability of a collision – two blocks having the same signature – is minimal. Prop. 3 shows that algebraic signatures allow us to update a signature of a changed block without recalculating the signature of the whole block. Without this property, maintaining signatures of large blocks through updates would involve recalculations. Prop. 5 implies that we can use our algebraic signature in order to check the coherency of data blocks with a parity block generated as the XOR of the data blocks. This proposition allows us to check the coherence of an xor parity block with its data blocks through signature comparisons alone. We expand this key property in Section 5.

4.3. Signature Calculation

If we directly implement the definition, calculating the signature of a block with n symbols takes ~2n multiplications and n additions. A typical block in our scheme might consist of 16M symbols of two bytes each, though larger blocks seem practical. Because of this large size, it is important to improve the signature calculation. We can do this by picking a special primitive element α, namely the bit string 0000…0010. We can multiply and element (bit string) β by α by shifting β to the left. If this results in an overflow, we XOR with a constant bit string, the so-called generator polynomial of the Galois field. Since we can use a Horner scheme, that is, write

$$\mathrm{sig}_\alpha(P) = \Big(\big(\big(\big(p_l \cdot \alpha + p_{l-1}\big)\alpha + p_{l-2}\big)\alpha \cdots + p_2\big)\alpha\Big) + p_1$$

we have reduced the complexity of the signature calculation considerably. To process a single symbol in the block, we multiply the signature up to this symbol with α by shifting and reducing if necessary, and then xoring the symbol to the current signature. Following [2], we can further improve the multiplications by α by accumulating the overflows of several multiplications and reduce them according to a pre-computed table. Processing a single element costs us then a shift and an XOR and every so often, processing the overflow bits by a table look-up and a xoring. This technique also applies to α^2 = 0000 … 0100, α^3 = 0000 … 1000 and other small powers of α, though with less savings.

5. Erasure Correcting Codes and Signatures

To generate the parity blocks from the data blocks in a redundancy group, we use an erasure correcting code. Given a vector $\boldsymbol{a}=(a_1, a_2, \ldots, a_m)$ of *data symbols*, where a_1 is taken from the first data block, a_2 from the second data block, etc., an ECC generates a longer vector $\boldsymbol{u} = (a_1, a_2, \ldots, a_m, a_{m+1}, \ldots, a_{m+k})$ where the *parity symbols* a_{m+i} are calculated from the data symbols. A block

consists of a string of symbols. We use the first symbol in all data blocks to form the data symbol vector $\boldsymbol{a}$, then use the ECC to generate vector $\boldsymbol{u}$, then take the $m+i^{\text{th}}$ coordinate of $\mathbf{u}$ to be the first symbol in the i^{th} parity block. We then use the second symbol in each data block to calculate the second symbol in each parity block, and so on, until we have populated the parity blocks.

The simplest ECC are n-fold replication codes, where we have only one data symbol and n-1 parity symbols, all equal to the data symbol. Formally, $\boldsymbol{a}=(a)$ and $\boldsymbol{u} = (a, a, \dots , a)$. The parity and the data blocks are then undistinguishable. The next simplest example is the m-parity code, with $\boldsymbol{a} = (a_1,\dots,a_m)$ and $\boldsymbol{u} = (a_1,a_2, \dots, a_m, a_1\oplus a_2\oplus\dots\oplus a_m)$. This is the RAID 5 scheme. Generalized Reed-Solomon codes use a generator matrix $\mathbf{G}$ with n rows and m columns. The generator matrix has the form $(\mathbf{I} \mid \mathbf{P})$ where $\mathbf{I}$ is the m-dimensional identity matrix. The block symbols and the coefficients are elements in the same Galois field. The relationship between the $\boldsymbol{a}$ and $\boldsymbol{u}$-vectors is simply $\boldsymbol{u} = \boldsymbol{a}\cdot\mathbf{G}$. The peculiar form of $\mathbf{G}$ implies that the coordinates of $\boldsymbol{a}$ are the first m coordinates of $\boldsymbol{u}$. To calculate a single parity symbol, we can merely multiply $\boldsymbol{a}$ with the corresponding column of $\mathbf{P}$. The m-parity code is a special case with $\mathbf{P} = {}^{\text{t}}(1, 1, \dots ,1)$. (The superscript "t" stands for transpose, that is, $\mathbf{P}$ is a m-dimensional column vectors with m coefficients 1.) The following proposition generalizes Prop. 5:

Proposition 6: Let $B_1, \dots B_m$ be m data block and let $B_{m+1}, \dots B_n$ be the parity blocks calculated with a generalized Reed-Solomon code. Then $\left(\text{sig}_\beta(B_1),\text{sig}_\beta(B_2),\dots,\text{sig}_\beta(B_m),\text{sig}_\beta(B_{m+1}),\dots,\text{sig}_\beta(B_n)\right)$ is a code word of the generalized Reed-Solomon code.

Prop. 6 says that the signature of a parity block is calculated as the Reed-Solomon code calculated parity of the signatures of the data blocks. Accordingly, by looking at the signatures of the blocks only, we can discover (with very low probability of error) whether the data blocks indeed generated these parity blocks. If $k = n$-m is large enough to correct errors, we can even determine which data block has been changed without changes to the parity blocks or which parity blocks have not been updated, as long as the number of blocks in error is smaller than $k/2$.

Proof: Let P_λ, $\lambda \in \{m+1, \dots, n\}$ be a parity block. Denote the i^{th} element of P_λ with $p_{i,\lambda}$. Similarly, denote the i^{th} element of a data block P_ν with $p_{i,\nu}$. Finally, write the coefficient in row i and column j of the generator matrix $\mathbf{G}$ as $g_{i,j}$. According to the definition of generalized Reed-Solomon codes,

$$p_{i,\lambda} = \sum_{\nu=1}^{m} p_{i,\nu} g_{\nu,\lambda}$$

We now calculate

$$\begin{aligned}
\text{sig}_\beta(P_\lambda) &= \sum_{i=1}^{l} \beta^{i-1} p_{i,\lambda} \\
&= \sum_{i=1}^{l} \beta^{i-1} \sum_{\nu=1}^{m} p_{i,\nu} g_{\nu,\lambda} \\
&= \sum_{i=1}^{l} \sum_{\nu=1}^{m} \beta^{i-1} p_{i,\nu} g_{\nu,\lambda} \\
&= \sum_{\nu=1}^{m} \sum_{i=1}^{l} \beta^{i-1} p_{i,\nu} g_{\nu,\lambda} \\
&= \sum_{\nu=1}^{m} (\sum_{i=1}^{l} \beta^{i-1} p_{i,\nu}) g_{\nu,\lambda} \\
&= \sum_{\nu=1}^{m} \text{sig}_\beta(P_\nu) g_{\nu,\lambda}
\end{aligned}$$

Thus, the signature of the parity block is calculated as the Reed-Solomon code parity of the signatures of the data blocks and the proposition follows.

Convolutional Array Codes (CAC) use only XOR operations to generate parity data. If we think of the bits in the data block put into columns, then a CAC uses horizontal and diagonal parity lines at various slopes to generate parity bits in additional parity columns. A CAC will generate parity blocks that are longer than data blocks. However, this "overhang" is only a few bytes and could be stored together with metadata of the parity block. Figure 1 shows a small CAC with three data blocks and three parity blocks. The first parity block (block 4 from left to right) contains the parity along horizontal lines, the second one along lines of slope 1, and the last one along lines of slope 2. Since we can store the overhang separately from the blocks with the metadata, CAC are very attractive because of their speed and simplicity of erasure correction. We have the analogue to Proposition 6.

Proposition 7: Let P be a parity bucket generated by a CAC along the lines of slope s. Let $B_1, \dots, B_m$ be the data buckets. Then $\text{sig}_\beta(P) = \text{sig}_\beta(B_1) + \beta^s \text{sig}_\beta(B_2) + \dots + \beta^{(m-1)s} \text{sig}_\beta(B_m)$.

Proof: We recall that the addition in the Galois field is the XOR operation. The symbols p_i in P are thus sums of symbols in $B_1, \dots, B_m$. We write $p_{i,\lambda}$ for the i^{th} symbol in B_λ and p_i for the i^{th} symbol in P. The number

of symbols in P is $l+(m-1)s$. In the following formulae, we assume that we pad formally the data blocks B_λ with zeroes, that is, that $p_{i,\lambda} = 0$ if $i<1$. We have

$$p_i = \sum_{\nu=1}^{m} p_{i-(\nu-1)s,\nu} \ .$$

We now calculate

$$\begin{aligned}
\mathrm{sig}_\beta(P) &= \sum_{\mu=1}^{l+(m-1)s} \beta^{\mu-1} p_\mu \\
&= \sum_{\mu=1}^{l+(m-1)s} \beta^{\mu-1} \sum_{\nu=1}^{m} p_{\mu-(\nu-1)s,\nu} \\
&= \sum_{\nu=1}^{m} \sum_{\mu=1}^{l+(m-1)s} \beta^{\mu-1} p_{\mu-(\nu-1)s,\nu} \\
&= \sum_{\nu=1}^{m} \beta^{(\nu-1)s} \sum_{\mu=1}^{l} \beta^{\mu-1} p_{\mu,\nu} \\
&= \sum_{\nu=1}^{m} \beta^{(\nu-1)s} \mathrm{sig}_\beta(B_\nu)
\end{aligned}$$

This proves the proposition.

					a_0
					a_1
				a_0	$a_2 \oplus b_0$
				$a_1 \oplus b_0$	$a_3 \oplus b_1$
a_0	b_0	c_0	$a_0 \oplus b_0 \oplus c_0$	$a_2 \oplus b_1 \oplus c_0$	$a_4 \oplus b_2 \oplus c_0$
a_1	b_1	c_1	$a_1 \oplus b_1 \oplus c_1$	$a_3 \oplus b_2 \oplus c_1$	$a_5 \oplus b_3 \oplus c_1$
a_2	b_2	c_2	$a_2 \oplus b_2 \oplus c_2$	$a_4 \oplus b_3 \oplus c_2$	$a_6 \oplus b_4 \oplus c_2$
a_3	b_3	c_3	$a_3 \oplus b_3 \oplus c_3$	$a_5 \oplus b_4 \oplus c_3$	$a_7 \oplus b_5 \oplus c_3$

Figure 1: Array code example

6. Signature Data Structure

We recall that large storage systems place data into redundancy groups. These consists of n blocks so that $m < n$ suffice to rebuild all data on these blocks. As we have seen, if the data in the redundancy group do not reflect the same state of the system, that is, if the redundancy group lacks coherence, then the capacity of the system to recover from device failure might be lost. Ascertaining redundancy group coherence is the goal of our scheme. Our solution is build on another need, namely to quickly detect bit rot and other types of masked, partial failure of a device.

The size of the blocks used for redundancy group is usually much larger than the 512B of disk blocks or the small multiples of these blocks in which file systems typically allocate storage. The results in [18] and [19] indicate that currently block sizes of GB range are appropriate. When we are speaking of blocks here, we mean these larger blocks.

We maintain a single data structure that maintains the m-signature with base α of each block of a redundancy group on the disk drive. When we change data in the block, we calculate the new signature from the old signature value and the signature of the change, by applying Prop. 3 to the coordinates of the m-signature. In this way, maintaining the signature map only costs minimal overhead.

We run two distributed background processes in order to find masked failures, including redundancy group incoherence. First, we periodically scrub a disk, that is, if necessary (such as in a MAID [4], [5]), we power a device up and compare the m-signatures maintained in our data structured with the one recalculated by accessing all the blocks. This operation detects media defects and data corruption. Second, we periodically use Prop. 5, Prop. 6, or Prop. 7 to check whether the m-signatures of the parity blocks are the ones that the signatures of the data blocks in the redundancy group imply. If this is not the case, then we know that the data in the redundancy group is incoherent. If we use replication instead, we merely have to compare the signatures. Detecting this problem before a device failure triggers a reconstruction gives us more leverage to fix the problem, if necessary by simply recalculating parity blocks.

7. Related Work

Signatures are frequently used to identify objects and to capture changes. Applications include pattern-matching (e.g. the fundamental proposal by Rabin and Karp [10]), protection against unauthorized updates (e.g. Tripwire by Kim and Spafford [11]), discovering differences in replicated databases, first proposed by Metzner [15] but then elaborated by many authors), synchronizing replicas (e.g. the work by Suel, Noel, and Trendafilov [17]), and distributed backup such as Pastiche by Cox, Murray, and Noble [6], to name but a few. Some applications use signatures with algebraic properties, e.g. decomposable hash functions [17]. Karp–Rabin fingerprinting [10] uses a formula close to the one that we are proposing for a distributed pattern matching algorithm that only sends the signature of the pattern to the searched sites. Karp-Rabin type fingerprints have recently been used to discover

similarity in documents, e.g. [2]. To my best knowledge, no one has yet used signatures with algebraic properties to compare parity data. Litwin and Schwarz [14] derive the fundamental properties of algebraic signatures and discuss their use in Scalable Distributed Data Structures (SDDS). Litwin, Mokadem, and Schwarz [12] report on a first implementation of the signature calculations. The reported speeds show that algebraic signature calculation is about as fast as that of SHA1 signatures, but we believe that the results can be improved.

8. Conclusion and Future Work

As storage systems grow in size, statistically burst of failures become likely. Large size (PB-scale) and increasing demands on resilience force us to take failure modes into account that are negligible for smaller systems. This paper addresses one such mode, namely the undetected discrepancy between client data and parity data in a redundancy group. This paper proposes a solution that works in conjunction with the detection of block failures such as media defects. I have presented an algebraic signature that can discover whether parity data generated by XORing, by a generalized Reed-Solomon code, or by a convolutional array code reflects faithfully the client data.

Future work will include an implementation to prove the computational feasibility of the scheme. We also need to integrate disk scrubbing with the design of large scale storage systems, evaluate its impact on system reliability, and investigate optimal strategies in systems where disks are typically powered off (MAIDs).

Acknowledgement

I gratefully acknowledge support from the Santa Clara University IBM Research Grant EIBM0015 and from a generous gift by Microsoft Research.

References

[1] Blaum, M., Farrell, G., van Tilborg, H.: Array codes. In Pless, Huffman (ed.) Handbook of Coding Theory II, p. 1855 – 1909. North Holland, 1998.

[2] Broder, A. Some applications of Rabin's fingerprinting method. In: Capocelli, De Santis, and Vaccaro, (ed.), Sequences II: Methods in Communications, Security, and Computer Science, p. 143 – 152. Springer-Verlag, 1993.

[3] Corbett, P, English, B., Goel, A., Grcanac, T., Kleiman, S., Leong, J. and Sankar, S.: Row-diagonal parity for double disk failure correction. In *Proc. of 3d Usenix Conf. on File and Storage Technologies, San Francisco, CA,* 2004.

[4] Colarelli, D. and Grunwald, D.: Massive arrays of idle disks for storage archives. In *Proc. IEEE/ACM Conf. on Supercomputing (SC2002)*, p. 47-58. 2002.

[5] Colarelli, D., Grunwald, D., and Neufeld, M.: The case of massive arrays of idle disks (MAID). In *Proceedings of Usenix FAST '02*. 2002.

[6] Cox, L., Murray, C., and Noble, B.: Pastiche: making backup cheap and easy. In *Proc. of 5th Sym. Operating Systems Design and Implementation, OSDI'02*, p. 285-298. 2002.

[7] Elerath, J. and Shah, S.: Disk drive reliability case study: Dependence upon head fly-height and quantity of heads. In *2003 Proc. Annual Reliability and Maintainability Symposium,* RAMS'03, p. 608-612. 2003.

[8] Hellerstein, L, Gibson, G., Karp, R., Katz, R. and Patterson, D.: Coding techniques for handling failures in large disk arrays. In *Algorithmica*, vol. 12, p. 182-208, 1994.

[9] Internet Archive. http://www.archive.org/web/researcher/data_available.php.

[10] Karp, R. and Rabin, M.: Efficient randomized pattern-matching algorithms. In *IBM Journal of Research and Development*, Vol. 31, No. 2, March 1987.

[11] Kim, G. and Spafford. E.: The Design and implementation of Tripwire: A file system integrity checker. In *Proc. of the ACM Conference on Computer and Communications Security*, p. 18-29, 1994.

[12] Litwin, W., Mokadem, R. and Schwarz, T.: Disk backup through algebraic signatures in scalable and distributed data structures. *In Proc. 5th Workshop on Distributed Data and Structures, Thessaloniki*, 2003 (WDAS'03).

[13] Litwin, W., Schwarz, T.: LH*$_{RS}$: A High-Availability Scalable Distributed Data Structure using Reed-Solomon Codes. In *Proc. 2000 ACM SIGMOD Int. Conf. on Management of Data,* Dallas 2000, p. 237-247.

[14] Litwin, W., Schwarz, T. Algebraic Signatures for Scalable Distributed Data Structures. *Proc. of the 20th International Conference on Data Engineering (ICDE),* Boston, 2004, p. 412-423.

[15] Metzner, J. *A Parity Structure for Large Remotely Located Data Files*. IEEE Transactions on Computers, Vol. C – 32, No. 8, 1983.

[16] Schwarz, T., Bowdidge, R. and Burkhard, W.: Low Cost Comparison of File Copies. In Proc. Intern. Conf. on Distributed Computing Systems, Paris, Fr., 1990, (ICDCS 5 Proceedings), p. 196-202.

[17] Suel, T., Noel, P., and Trendafilov, D.: Improved File Synchronization for Maintaining Large Replicated Collections over Slow Networks. In *Proc. 20th Int. Conf. on Data Engineering, ICDE*, Boston, 2004, p. 153-164.

[18] Xin, Q., Miller, E, Long, D., Brandt, S., Litwin, W., and Schwarz, T. Selecting reliability mechanisms for a large object-based storage system. In *20th Symp. on Mass Storage Systems and Technology.* San Diego. 2003.

[19] Xin, Q., Miller, E, Schwarz, T: Evaluation of distributed recovery in large-scale storage systems. *13th IEEE*

International Symposium on High Performance Distributed Computing, HPDC'04, Honolulu, HI, 2004.