DOC: T10/03-274r0

Use of non-XOR/CRC Guards in RAID Applications

From: Walter Rassbach

This document describes the advantages and extra protection provided by a non-XOR/CRC based guard calculation method in a RAID (or RAID-like) context. This document discusses a particulat checksum based block guard calculation method but but the overall discussion actually applies to any other block guard calculation method which is no based on XOR (or CRC).

The document primarily discusses a set of simplified, fairly concrete example because the crux of the claimed advantage is the result of the way the actual calculations work out. The calclations in the general case work out similarly due to the mathematical relationships. In particular, the fact that an XOR/CRC-based guard is subject to the "corner-sum" characteristics described is the result of the mathematical associativity and commutivity of XOR while the claimed advantages of the checksum method (or other non-XOR/CRC methods) arise directly from the fact that those mathematical properties do not hold between XOR and "addition".

First, it must be recognized that a CRC (against a fixed size block) is really just a fancy way of specifying a set of XOR trees. Each bit in the CRC-value calculated for a block of data is simply the XOR of a selected subset of the bits in the data block. Looking at it from the other side, each bit position in the data is associated with a specific signature in the CRC field and the CRC-value calculated for a given block of data is simply the XOR of the signatures of each active data bit, with the seed-signature (if the calculation is seeded) which consists of the seed-value "clocked through" the CRC the appropriate number of times. In particular, the following relationship holds for any two blocks of data, A and B:

$$crc(A) \oplus crc(B) = crc(A \oplus B) \oplus \text{<seed-signature>}$$

The "extra" seed-signature on the right-hand side is due to the fact that both terms on the left-hand side contain the seed-signature. For simplicity in the remainder of the discussion, we will presume that the seed (and thus, the seed-signature) is zero -- The adjustments to deal with non-zero seeds are straightforward. In addition, for ease of presentation, we will use a simple LRC (which is really just a CRC using a polynomial of the form $x^{16} + 1$) in the examples -- The examples are valid with any form of CRC, the calculations are just harder to follow and verify.

We will assume that we are working with a RAID-5 (or RAID-4) stripe consisting of data blocks A, B, C, and D, and parity block P. Each data block consists of the actual data (e.g., dataA), which is nominally 512 bytes long, and a check value (e.g., chkA) which is 2 bytes and is calculated from dataA. Both fields are saved on the media. The data part of the parity block, P, is simply the XOR of the data parts of the data blocks, i.e., $\text{dataA} \oplus \ldots \oplus \text{dataD}$. The goal of this discussion is to describe a method of calculating the check field value, chkP, to be stored on the media with the parity data, dataP, in such a way as to provide certain protections for the whole stripe.

If an XOR-based guard calculation method (e.g., LRC or CRC) is used for the data guard on the data blocks, the guard value calculated for the parity block can be taken to be either the guard value calculated directly from the parity block or it can be the XOR of the guard values of the data blocks -- The two values differ by a fixed amount which can be determined from the seeding method and does not depend on any of the actual data. The reason that this occurs is that all of the (actual) operations involved use XOR and XOR is both commutative and associative. If the value

used to seed the guard calculations is zero or the seeding method is designed appropriately, the difference between the two possible choices can be eliminated and the method used to calculate the check-value for the parity block does not matter. In this case, the check-value for the parity block is essentially equivalent to a corner-sum used to cross-check row-and-column additions. This is illustrated in the following example, which uses LRC, no seeding, and 4-bit "words":

	blk A	blk B	blk C	blk D	Parity
nib 0	9	4	8	C	9
nib 1	0	F	A	3	6
nib 2	8	F	B	2	E
nib 3	6	1	0	7	0
nib 4	1	5	D	4	D
LRC	6	0	4	E	C

Initial "stripe" using an LRC

The LRC/Parity value can be obtained by XORing by rows then columns or columns then rows. If an update-write to (say) block A of the stripe is done, the old value of block A and the old parity block are read, the new parity block is constructed, and both the new block A and the new parity block are written to disk. For purposes of the discussion, we will ignore the possibility of any independent recovery methods (e.g., based on non-volatile memory or transaction logging).

If, during an update-write, a failure (e.g., loss of power) occurs where one of the new blocks is written but the other is not (often called a "write hole"), the stripe is no longer consistent since the parity block does not match the other blocks in the stripe. There are two cases:

1. The new block A is written but the parity is not. In this case, a read of block A will normally fetch the new value, but a reconstruction will result in the old value.
2. If the parity block is written but the new value of block A is not written, a read of block A will normally get the old value of block A but a reconstruction will result in the new value.

Most good RAID implementations provide mechanisms to locate and repair such inconsistent stripes. Following the repair, the value stored for block A may be either the old value or the new value, but this is generally not an issue since (assuming that there is no fast or cache write policy) status would not have been presented for the failed write. Moreover, if either the drive containing block A or the parity block fails before the repair, there is still no problem since the recovery mechanisms will produce either the old or new value for block A.

However, there is a problem if the drive containing block B should fail before the repair is made, and this problem is severe because block B was not supposed to be affected or at risk during the failed update-write operation. Moreover, unless the implementation provides some 'sideband" mechanism for remembering that the stripe is at risk, it does not have any way of detecting the inconsistent stripe if it uses an XOR-based guard calculation method like a CRC, because the value reconstructed for block B from the inconsistent stripe will always have a valid guard value (after any adjustments due to the seeding methods). The result, in most implementation, is that block B will be randomly altered and the implementation will present it as "good data".

Note that this is a direct mathematical consequence of the use of an XOR-based guard calculation method, such as an LRC or CRC. Such an implementation simply cannot detect this problem by the use of the block guards, and "sideband" methods for detecting such problems are generally expensive and/or complex and error-prone.

For example, using the stripe above, if the new value for block A is 4, A, 8, 7, 3, with a guard value of 2, and the new block A is written but the parity block is not, the reconstruction is:

	blk A	[blk B]	blk C	blk D	Parity	blk B
nib 0	4	[4]	8	C	9	9
nib 1	A	[F]	A	3	6	5
nib 2	8	[F]	B	2	E	F
nib 3	7	[1]	0	7	0	0
nib 4	3	[5]	D	4	D	7
LRC	2	[0]	4	E	C	4

Block A updated, Parity not updated, Block B rebuilt incorrectly (on right) using an LRC

If the parity is written but block A is not written, the reconstruction is:

	blk A	[blk B]	blk C	blk D	Parity	blk B
nib 0	9	[4]	8	C	4	9
nib 1	0	[F]	A	3	C	5
nib 2	8	[F]	B	2	E	F
nib 3	6	[1]	0	7	1	0
nib 4	1	[5]	D	4	F	7
LRC	6	[0]	4	E	8	4

Block A not updated, Parity updated, Block B rebuilt incorrectly (on right) using an LRC

Note that the reconstructed block B is the same in both instances -- This is because the value which results from the reconstruction is the correct value for block B XORed with the old and new values for block A. The reconstructed guard value for the erroneously reconstructed block B corresponds to the reconstructed data due to corner-sum characteristics and no error is detected.

If a check-sum based (or some other non-XOR-based) guard calculation method is used, the reconstructed block will generally fail to correspond to the reconstructed data. This holds because addition and XOR are incompatible operations and the corner-sum characteristics are eliminated. If a check-sum based guard method is used, the check value stored with the parity block should be the XOR of the check values for the data blocks and not the check-sum of the parity block. This means that the parity block cannot be independently validated (since its guard value generally is not the value calculated using the check-sum based method against the parity block), but, in return, it provides a “synchronization” check across the whole stripe.

To extend the examples above, the check-sum based guard will use a simple 4-bit “word” 1’s compliment check-sum (i.e., a carry from the top bit will be wrapped to the bottom bit), with a seed value of F (equivalent to a 0, but this ensures that the calculated value will never be 0) -- In actual practice, a slightly more complicated method (described below) should be used to provide the guard with certain additional desirable characteristics. The initial stripe would be:

	blk A	blk B	blk C	blk D	Parity
nib 0	9	4	8	C	9
nib 1	0	F	A	3	6
nib 2	8	F	B	2	E
nib 3	6	1	0	7	0
nib 4	1	5	D	4	D
sum	9	A	C	D	2

Initial “stripe” using a checksum (4 bt wrapped carry)

Note that the value of the check-sum of the parity block is C, but the value stored with the parity block is the XOR of the guard values for the data blocks, i.e., $9 \oplus A \oplus C \oplus D$.

The guard value for the new block A (4,A,8,7,3) works out as 2 again. In the case where block A is written but the parity block is not written, the reconstruction example is:

	blk A	[blk B]	blk C	blk D	Parity	blk B
nib 0	4	[4]	8	C	9	9
nib 1	A	[F]	A	3	6	5
nib 2	8	[F]	B	2	E	F
nib 3	7	[1]	0	7	0	0
nib 4	3	[5]	D	4	D	7
sum	2	[A]	C	D	2	1 / 8

Block A updated, Parity not updated, Block B rebuilt incorrectly (on right) using a checksum The checksum does not crosscheck!

Where the reconstructed guard value is $2 \oplus C \oplus D \oplus 2$ which works out to 1, while the guard value calculated for the reconstructed block is 8. As a result, this block will fail the guard check and be recognized as an invalid block. The implementation may not be able to determine the exact cause of the failure, but it will at least be able to avoid presenting incorrect data as if it were the correct and valid data.

In the second case, where the parity block is written but the new value for block A is not written, the reconstruction example is similar:

	blk A	[blk B]	blk C	blk D	Parity	blk B
nib 0	9	[4]	8	C	4	9
nib 1	0	[F]	A	3	C	5
nib 2	8	[F]	B	2	E	F
nib 3	6	[1]	0	7	1	0
nib 4	1	[5]	D	4	F	7
sum	9	[A]	C	D	9	1 / 8

Block A not updated, Parity updated, Block B rebuilt incorrectly (on right) using a checksum The checksum does not crosscheck!

The guard value stored in the updated parity block reflects the guard value for the new block A and is $2 \oplus A \oplus C \oplus D$ which works out to 9 (the actual check-sum for the new parity block is 1). Once again, the reconstructed block will fail the guard checks and be recognized as invalid.

As these examples show, an XOR based guard does not provide any kind of protection across a stripe because of the corner-sum characteristics while using a check-sum based guard calculation method with the XOR of the data block guards stored with the parity block (rather than using the check-sum calculation of the parity block itself) provides a “synchonrization” check across the stripe during a reconstruction operation.

This check is essentially as strong as the guard itself, e.g., if the guard is 16 bits (i.e., a 1-in-2^{16} strength), the cross stripe “synchronization check” strength will also be 1-in-2^{16}.

A check-sum based calculation should use 1’s compliment addition (i.e., carries from the top bit of the accumulation should be wrapped to the bottom bit). This ensures that all bits in the guarded data are treated similarly rather than providing essentially stronger protection for the low order bits which would be the result of not wrapping the carries. In addition, if the calculation is seeded

with a non-zero value (all 1's is a possible choice, which has the same effect as a zero seed), the resulting value will always be non-zero. This allows a value of zero to be used as a special "marker" value when stored in place of the normal guard value.

Mathematically, such a check-sum is quite similar to what a CRC is. In the check-sum case the value calculated is the remainder of a division of the data block as a single (large) integer by a value of 2^{16}-1. In the case of a CRC, the value calculated is the remainder of a division of the data block, as a polynomial over GF(2), by the CRC polynomial.

However, a CRC provides protection against certain types of swaps and displacements of the data that the basic check-sum does not provide. A check-sum based guard (even with rotation) is also subject to some "pattern errors" involving changes to only a few bits and/or bits that are not very far appart (short span errors). To provide such protection characteristics with a check-sum based guard, certain adjustments must be built into the calculation.

One basic adjustment is to multiply the accumulated value by 2 (using 1's complement arithmetic) at each step of the calculation -- Due to the use of 1's complement arithmetic, this amount to a left rotation of 1 bit. If the guard is 16 bits, this provides protection against displacements or swaps which are not a multiple of 32 bytes in distance. Providing protection against such swaps or displacement distances is a bit harder but not difficult: Inserting a multiplication by a factor of 7 every eighth step will protect against such displacements (Note: The multiplier here should not be a divisor of 2^{16}-1. Applying it every eighth step simplifies calculations in code and correlates with 16 byte boundaries). Alternatively, a uniform multiply by 7 modulo 2^{16}-1 every step (rather than multiplying by 2 on most steps) would protect against swaps or displacements that are not a step distance multiple of 512 bytes.

A simple multiply-bu-2 approach is also suject to some pattern errors. There are two main cases that could be worrisome:

1. Cases where one bit is flipped from 0 to 1 and another bit is flipped from 1 to 0, with the bits separated by a particular distance. The distance is generally a multiple of 17 bits, but it may be adjacent bits if they are in separate (adjacent) 16-bit "words".
2. An inversion of an aligned 16-bit word from an all 0's value to an all 1's value or vice-versa.

There are ways to address these problems, although they make the algorithms more complicated. The use of a (properly selected) multiplier other than 2 (modulo 2^{16}-1) will eliminate the cases where a change in just two bits (case 1 above) is undetected. For example, using a multiplier of 7 ensures that a minimum of 4 bits-in-error is required before similar cases occur. Other multipliers can increase the required number of bit-in-error slightly. Addressing the second problem (case 2) is somewhat more difficult. Simply XORing in a constant 16-bit value to each data word before adding in each word of data will shift the problem situation to a different pair of values.

However, there is a question about whether such measures are required in this particular context. In actual usages, the data blocks are generally already protected against such errors by other mechanisms, e.g., disk ECC, memory parity or ECC, or packet CRCs on serial trnasmission links. In such contexts, the correct question to be asking is: "whether errors undetected by such methods are <u>also</u> undetected by the guard". In other words, the question of the kinds of pattern error that are an issue cannot be independently examined, it must be examined in the context of any other protection mechanism that is in use.

The suggested code to calculate the suggested check-sum based guard is as follows:

```
register u32 accum;
register u16 *d_buf_p;
register int ctr;

accum = 0xFFFF;
d_buf_p = buffer_p;
ctr = 256;

do {
    accum <<= 1;
    accum +=  *d_buf_p++;
    if( (--ctr) & 7)
        continue;

    /* Optionally multiply by 7 here:
     accum = (accum <<3) - accum;
    */

    /* Wrap carries for the last 8 adds: */
    accum = (accum & 0xFFFF)
          + (accum >> 16);

    if( !ctr )
        break;
   } while( TRUE );

/* Finish carry wrapping: */
accum = (accum & 0xFFFF)
      + (accum >> 16);
```

This is the calculation for a 512 byte block. The algorithm can easily be extended to deal with blocks of arbitrary size. A guard value calculated this way provides essentially the same level of protection provided by a CRC and also provides a guard mechanism that provides the cross stripe “synchronization/consistency” check capability. The computation method can be elaborated in hardware to deal with the data in larger “chunks” that 2 bytes at a time. In code, this calculation is more than twice as fast as a calculation of a CRC.

For protection equivalent to an LRC, the simple wrapped carry check-sum can be used. The value can be accumulated in a 32-bit register with the carries all wrapped at the end. this is essentially as fast as the LRC calculation.