Binary Linear ECCs Optimized for Bit Inversion in Memories with Asymmetric Error Probabilities

Valentin Gherman†, Samuel Evain†, Bastien Giraud‡

†Paris-Saclay Campus, Nano-INNOV
91191 Gif sur Yvette, France

‡CEA-Leti, Minatec Campus,
Grenoble, France

***Abstract*—Many memory types are asymmetric with respect to the error vulnerability of stored 0's and 1's. For instance, DRAM, STT-MRAM and NAND flash memories may suffer from asymmetric error rates. A recently proposed error-protection scheme consists in the inversion of the memory words with too many vulnerable values before they are stored in an asymmetric memory. In this paper, a method is proposed for the optimization of systematic binary linear block error-correcting codes in order to maximize their impact when combined with memory word inversion.**

Keywords—memory word inversion; asymmetric error rates

I. Introduction

The majority of memory technologies have strong asymmetries with respect to the error rates that affect stored 0's and 1's. For instance, in DDR4 memories, the difference between the error rates may go up to 2 decades at 30°C and 4 decades at 60°C [6]. In 2-bit NAND flash memories, the first bit programmed in a storage cell is more vulnerable to retention errors when it is equal to 0 [1]. In STT-MRAM memories, stored 1's are more vulnerable to write and retention errors than stored 0's. The resulting error rate difference may reach 3 orders of magnitude [3][8].

A mitigation technique used to address the error rate asymmetry consists in the inversion before storage, via the logical not operator, of the memory words with too many vulnerable values. This solution enables significant error rate reductions despite the addition of one bit per memory word to indicate its inversion state [6].

This paper is focused on the optimal selection of error-correcting codes (ECC) in conjunction with memory word inversion. The objective is the evaluation of the uncorrectable error rate improvement and the impact of considering the vulnerable values among the check-bits when taking the decision of inverting a code word.

II. Systematic Binary Linear Block ECCs

Linear block ECCs can be defined with the help of a *parity-check matrix* or *H-matrix*. Any code word v of an ECC should satisfy the following relation [7]:

$$H \cdot v^T = 0 \tag{1}$$

where v^T is a column vector. In binary ECCs, the H-matrix and code words contain only binary values.

Error correction and detection properties are ensured via an appropriate selection of the H-matrix columns. For example, single-error correction is enabled if each H-matrix column is unique and different from the all-0 vector [4]. Double-error detection can be achieved if an additional check-bit is used to impose a fixed parity to all code words [2].

In systematic ECCs, one can make the distinction between data-bits and check-bits. The H-matrix of an ECC with k data-bits and r check-bits, can be structured as follows [2]:

$$H = [P, I_r] \tag{2}$$

where P is an $r \times k$ matrix and I_r is the $r \times r$ identity matrix.

Each line of the P-matrix can be used to compute one check-bit. For example, consider the H-matrix below:

$$H = \underbrace{\begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix}}_{P} \underbrace{\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}}_{I_3}$$

Any code word $v = (d_1, d_2, d_3, c_1, c_2, c_3)$ should satisfy (1) and its check-bits can be calculated as follows:

$$\begin{aligned} c_1 &= d_2 + d_3 \\ c_2 &= d_1 + d_3 \\ c_3 &= d_1 + d_2 + d_3 \end{aligned} \tag{3}$$

where the symbol '+' stands for the modulo-2 sum.

According to (3), if all data-bits are inverted the check-bits c_1 and c_2 will preserve their values while c_3 will be inverted. This is due to the fact that c_3 depends on an odd number of data-bits while c_1 and c_2 depend on an even number of data-bits. Generally, if all data-bits are inverted a check-bit is inverted if and only if it depends on an odd number of data-bits.

In the following, it will be assume that an extra bit is inserted in each code word to indicate its inversion state [6]. During the ECC encoding and decoding operations, the inversion bit is treated as a data-bit.

III. Code Word Inversion

Definition 1: In a systematic binary linear block ECC, a P-matrix line is called *odd (even)* if it contains an odd (even) number of entries equal to 1. Equivalently, an *odd (even)* check-bit depends on an odd (even) number of data-bits.

Definition 2: A binary ECC is called *inversion invariant* if the inversion of all bits in a code word results in another code word.

Theorem: A systematic binary linear block ECC is inversion invariant if and only if all its check-bits are odd. (An informal demonstration is given at the end of Section II.)

There are systematic binary linear block ECCs that cannot be made inversion invariant. For example, it is impossible to find an inversion invariant ECC with code words that have an odd number of bits and a fixed overall parity. This is due to the fact that inverting an odd number of bits will change their overall parity. It can be shown that a systematic binary linear block ECC with such properties contains at least one *even* check-bit. This results from the fixed overall parity of the code words that requires the existence of a linear combination of H-matrix lines equal to an all-1 vector [2]. This implies that at least one H-matrix line contributes to this linear combination with an odd number of 1's since the all-1 vector has an odd number of bits, just like the code words. Consequently, the systematic form of the H-matrix (2) will contain at least one line with an odd number of 1's which corresponds to an even P-matrix line and an even check-bit.

When inverting a code word that belongs to an ECC which is not inversion invariant, the even check-bits need to be kept unchanged in order to get another code word. In this way, the capability to perform error detection and correction on inverted words is preserved.

In order evaluate the impact of code word inversion, consider that the number of vulnerable values in a code word (cw) is decomposed as follows:

$$cw = \frac{k}{2} + i + l + j \tag{4}$$

where:

- k is the number of data-bits per code word except for the inversion bit,
- k is assumed to be an even number,
- $\frac{k}{2} + i$ represents the number of vulnerable values among the data-bits $\left(-\frac{k}{2} \le i \le \frac{k}{2}\right)$,
- l and j stand for the number of vulnerable values among the even and odd check-bits, respectively.

In (4), it is implicitly assumed that a non-inversion state of a code word is indicated by a non-vulnerable value assigned to the inversion bit. Recalling that the inverted version of a code word is obtained by inverting all bits with the exception of the even check-bits, the number of vulnerable values in the inverted version of a code word ($\overline{cw}$) can be decomposed as follows:

$$\overline{cw} = \frac{k}{2} - i + 1 + l + r - s - j \tag{5}$$

where:

- 1 stands for the vulnerable value taken by the bit used to indicate the code word inversion status,
- r represents the number of check-bits and s the number of even check-bits ($0 \le l \le s \le r$),
- r-s stands for the number of odd check-bits($0 \le j \le r$-s).

When the number of vulnerable values in a code word (cw) is larger than in its inverted version ($\overline{cw}$), the inverted version will be stored and the maximum number of vulnerable values per code word (max) can be computed as follows:

$$cw \ge \overline{cw} \tag{6}$$

$$i + j \ge \frac{r-s+1}{2} \tag{7}$$

$$max \le \frac{k+1+r+s}{2} \tag{8}$$

where (8) results from the combination of (5) and (7) and from the fact that l is smaller than or equal to s. No assumption is made on whether the vulnerable bit value is equal to 0 or 1. Similarly, it can be shown that (8) holds in the case when a code word has less vulnerable values than its inverted version. The same result is achieved if the non-inversion state of a code word is indicated by a vulnerable value assigned to the inversion bit.

According to (8), in the presence of selective code word inversion, the maximum (worst-case) number of vulnerable values per code word increases linearly with s i.e. the number of even check-bits. Consequently, among ECCs with the same error detection and correction capability, a maximum error rate improvement can be achieved by an ECC with a minimal number of even check-bits. In inversion invariant ECCs, the number of even check-bits is equal to zero. When the inversion invariance cannot be achieved, one should select an ECC with a single even check-bit. Our experience shows that it is relatively easy to find linear ECCs with single-error correction (SEC) or double-error correction (DEC) and a minimal number of even check-bits.

During a memory write operation, the decision of inverting a code word should be taken based on the evaluation of either (6) or (7). If one adds $\frac{k}{2}$ to both sides of (7) and discard the situation when the operands are equal, one obtains:

$$\frac{k}{2} + i + j > \frac{k+r-s+1}{2} \tag{9}$$

which means that the decision of inverting a code word can be taken based on the comparison between a constant and the number of vulnerable values among its data-bits and odd check-bits.

The evaluation of (9) can only be performed after the calculation of the odd check-bits and this may have an impact on the latency of memory write operations. This latency overhead can be avoided if the decision of inverting a code word is based on only the number of vulnerable values among the data-bits according to the expression below:

$$\frac{k}{2} + i > \frac{k+1}{2} \tag{10}$$

where the right-hand side is a constant.

According to (10), the code words in which $\frac{k}{2}$ data-bits are vulnerable do not have to be inverted. As the number of such code words is usually very high, in some of them all check-bits will have a vulnerable value. Accordingly, the maximum number of vulnerable values per code word (*max*) becomes:

$$max \leq \frac{k+2r}{2} \quad (11)$$

In the case of code words with fixed parity, the number of vulnerable values per code word is either even or odd and *max* should be even or odd, respectively. For example, if the vulnerable value is equal to 1 then *max* should be the largest even number that fulfills either (9) or (11).

IV. Simulation Results

This section is focused on the impact of considering the vulnerable values among the check-bits when taking the decision of inverting a code word. Only ECCs with a maximum number of odd check-bits are considered.

The first investigated metric is the uncorrectable bit error rate (UBER). When the stored 0's and 1's have different error rates, the UBER can be computed as follows [5]:

$$UBER = \frac{1}{k}\left[1 - \sum_{i=0}^{corr}\sum_{j=0}^{i}\binom{N_V}{j} RBER_V^j (1-RBER_V)^{N_V-j} \times \right.$$
$$\left. \times \binom{N_{nV}}{i-j} RBER_{nV}^{i-j} (1-RBER_{nV})^{N_{nV}-(i-j)}\right] \quad (12)$$

where:

- k is the number of data-bits per code word,
- *corr* stands for the maximum number of correctable errors per code word,
- *i* is iterated over the numbers of correctable erroneous bits in a code word,
- *j* and *i-j* represent numbers of correctable erroneous bits initially programmed to vulnerable and non-vulnerable values, respectively,
- N_V and N_{nV} stand for the numbers of vulnerable and non-vulnerable values in a code word,
- $RBER_V$ and $RBER_{nV}$ represent the raw bit error rates of the vulnerable and non-vulnerable values.

Each term of the sum operator used in (12) represents a combination of correctable errors among vulnerable and non-vulnerable values such that, finally, UBER is defined by the occurrence probability of all possible uncorrectable errors.

The worst-case UBER corresponds to the maximum number of vulnerable values per stored word. In the absence of any kind of inversion, all bits in a stored code word may be equal to a vulnerable value if (a) the vulnerable bit value is equal to 0 or (b) the ECC is inversion invariant. Both conditions are due to the fact that a linear ECC contains the all-0 code word. In the following, the less critical situation will be assumed in which the vulnerable bit value is equal to 1.

The reduction of the worst-case UBER achieved through the addition of an inversion bit and selective code word inversion is reported in the *inv* columns of Table I and Table II. The decision of inverting or not a code word is based only on the number of vulnerable values among the data-bits according to (10). The *inv+* columns give the additional reduction that can be obtained if the decision of inverting a code word is based on (9). The reduction reported in the *inv+* columns is calculated with respect to the UBER values obtained after the first reduction given in the *inv* columns.

As one might expect, the UBER reduction increases with the ratio between the RBERs of vulnerable and non-vulnerable values. The *inv+* reduction increases with the relative check-bit ratio in a code word which gets higher as (a) the error-correction strength increases and (b) the data-bit number per code word decreases. For 32-bit data words, selective code word inversion may reduce the worst-case UBER by up to 63% and 67% for single-error correction (SEC) and double-error correction (DEC) codes, respectively. With a maximum number of odd check-bits, the consideration of the vulnerable values among the check-bits enables an additional reduction of the worst-case UBER of 26% and 53%, respectively. No additional reduction would have been possible with ECCs characterized by a minimum number of odd check-bits.

The worst-case UBER is a pessimistic reliability metric as only code words with a maximum number of vulnerable values are taken into account. It is quite hard to imagine a software application that involves only such code words. Nevertheless, this metric has the merit of providing the maximum possible UBER reduction.

In order to get a more realistic evaluation of the potential reliability gains, we considered the impact on the mean UBER under the assumption that all possible code words of an ECC have the same storage probability. According to (12), two different code words may have different UBER values only if they have different numbers of vulnerable values. Therefore, in order to compute the UBER of a memory system under the assumption of uniform storage probability, it is sufficient to classify and count all code words according to the number of contained vulnerable values.

Unfortunately, for binary ECCs with more than 32 data-bits per code word, it may become infeasible to count one by one all code words with a given number of vulnerable values. In order to avoid this, a two-phase counting approach is used. In the first phase, all code words with at least 4 vulnerable or non-vulnerable values among the data-bits are counted one by one. This corresponds to the lowest counts of code words with a given number of vulnerable values among the data bits. The parameter 4 is a heuristic choice related to the minimum Hamming distance of the ECCs used here.

In a second phase, an approximate counting approach is used based on the probability of each check-bit to become equal to 1. For a given check-bit, this probability depends on (i) the number of 1's in the P-matrix line that is used to calculate it and (ii) the number of data-bits equal to 1. The check-bit values are assumed to be independent random variables,

wherefrom the approximate nature of this method. In the case of ECCs with fixed code word parity, a restriction is imposed on the allowed combinations of check-bit values for a given number of data-bits equal to 1. It appears that the obtained estimates fit very well to the exact counts.

The impact on the mean UBER of selective code word inversion is reported in Table III and Table IV for the same ECCs as in Table I and Table II. As before, the *inv+* columns give the additional reduction of the mean UBER that can be obtained if the decision of inverting a code word is based on (9). The improvement of the mean UBER increases with (a) the ratio between the RBERs of vulnerable and non-vulnerable values and (b) the check-bit ratio in the code words. For 32-bit data words, selective code word inversion may reduce the mean UBER by up to 20% and 26% for SEC and DEC codes, respectively. With an ECC that has a maximum number of odd check-bits, the consideration of the vulnerable values among the check- bits enables an additional reduction of 3% and 7%, respectively.

The encoders and decoders of the considered ECCs were implemented as combinational logic blocks sandwiched between two pipeline registers and synthesized with an ST 45nm standard cell library. In the case of the ECC decoders, the logic and latency overheads did not exceed 20%. The improvements reported in the *inv+* require no modification of the ECC decoder and implicitly 0% logic or latency overhead.

TABLE I. REDUCTION OF WORST-CASE UBER BASED FOR CODE WORDS WITH 32 DATA-BITS

$RBER_V$ / $RBER_{nV}$	SEC (39, 32+1)		SEC-DED (40, 32+1)		DEC (45, 32+1)		DEC-TED (46, 32+1)	
	inv	inv+	inv	inv+	inv	inv+	inv	inv+
10	57.7%	21.8%	57.5%	14.8%	61.2%	46.2%	61.1%	39.7%
10^2	62.8%	25.5%	62.8%	17.4%	66.3%	52.3%	66.3%	45.3%
10^3	63.3%	25.9%	63.3%	17.7%	66.8%	52.9%	66.8%	45.9%

TABLE II. SAME AS IN TABLE I FOR 64 DATA-BITS PER CODE WORD

$RBER_V$ / $RBER_{nV}$	SEC (72, 64+1)		SEC-DED (73, 64+1)		DEC (79, 64+1)		DEC-TED (80, 64+1)	
	inv	inv+	inv	inv+	inv	inv+	inv	inv+
10	62.9%	12.5%	63.2%	12.2%	71.4%	34.1%	71.3%	29.7%
10^2	67.9%	14.7%	67.2%	14.4%	76.1%	39.2%	76.1%	34.4%
10^3	68.4%	15.0%	67.6%	14.6%	76.5%	39.7%	76.5%	34.9%

TABLE III. REDUCTION OF MEAN UBER UNDER THE ASSUMPTION THAT ALL CODE WORDS HAVE THE SAME OCCURRENCE PROBABILITY FOR CODE WORDS WITH 32 DATA-BITS

$RBER_V$ / $RBER_{nV}$	SEC (39, 32+1)		SEC-DED (40, 32+1)		DEC (45, 32+1)		DEC-TED (46, 32+1)	
	inv	inv+	inv	inv+	inv	inv+	inv	inv+
10	15.0%	2.1%	14.6%	1.4%	19.7%	5.5%	19.3%	6.2%
10^2	19.1%	2.6%	18.6%	1.7%	25.0%	7.0%	24.4%	7.8%
10^3	19.5%	2.7%	19.0%	1.8%	25.6%	7.2%	25.0%	8.0%

TABLE IV. SAME AS IN TABLE III FOR 64 DATA-BITS PER CODE WORD

$RBER_V$ / $RBER_{nV}$	SEC (72, 64+1)		SEC-DED (73, 64+1)		DEC (79, 64+1)		DEC-TED (80, 64+1)	
	inv	inv+	inv	inv+	inv	inv+	inv	inv+
10	12.2%	0.8%	12.1%	0.8%	16.9%	2.5%	16.6%	2.1%
10^2	15.2%	1.0%	15.0%	0.9%	21.0%	3.2%	20.7%	2.7%
10^3	15.6%	1.0%	15.3%	1.0%	21.4%	3.3%	21.1%	2.8%

Concerning the ECC encoders, the logic overhead varied between 68% and 117% while the latency overhead was between 74% and 132%. The fact of counting the vulnerable values among the check-bits induced an additional logic overhead between 15% and 48% while the additional latency overhead varied between 3% and 32%. The encoders and decoders considered here are relatively small logic units that can be easily pipelined if the latency overhead becomes inacceptable.

V. CONCLUSIONS

A method is proposed for the optimization of systematic binary linear block ECCs in order to maximize their impact when combined with selective memory word inversion. This enables a reduction of the UBER when applied to memories that are asymmetric with respect to the error vulnerability of stored 0's and 1's. For example, in the case of 32-bit memories protected by a single-error correcting ECC without even check-bits, the worst-case UBER can be reduced by 26% if the check-bits are considered when taking the decision of inverting a code word. This improvement may reach 53% in the case of 32-bit memories protected by a double-error correction ECC. Similarly, under the assumption that all possible code words have the same storage probability, the mean UBER may be reduced by 3% and 7%, respectively. These improvements can be achieved without any storage overhead.

REFERENCES

[1] Y. Cai et al., "Error analysis and retention-aware error management for nand flash memory," Intel Technology Journal, Volume 17, Issue 1, pp. 140–164, 2013.

[2] C.L. Chen and M.Y. Hsiao, "Error-correcting codes for semiconductor memory applications: a state of the art review," Reliable Computer Systems - Design and Evaluation, Digital Press, 2nd edition, pp. 124–134, 1992.

[3] Y. Emre et al., "Enhancing the reliability of STT-RAM throught Circuit and System Level Techniques," IEEE Workshop on Signal Processing Systems, pp. 125–130, 2012.

[4] R.W. Hamming, "Error correcting and error detecting codes," Bell Sys. Tech. Journal, Vol. 29, April 1950, pp. 147-160.

[5] JEDEC Standard, "Solid-state drive (SSD) requirements and endurance test method," JESD218A, February 2011.

[6] K. Kraft, D.M. Mathew, C. Sudarshan, M. Jung, C. Weis, N. When and F. Longnos, "Efficient coding scheme for DDR4 memory subsystems," MEMSYS, pp. 148-157, 2018.

[7] S. Lin and D. J. Costello, "Error control coding: fundamentals and applications," Prentice-Hall, Inc., Englewood Cliffs, NJ, 1983.

[8] C. Yang et al., "Improving reliability of non-volatile memory technologies through circuit level techniques and error control coding," EURASIP Journal on Advances in Signal Processing, 2012:211, 2012.