This is a postprint version of the following published document:

Li, J., Reviriego, P., Xiao, L. & Wu, H. (2021). Protecting Memories against Soft Errors: The Case for Customizable Error Correction Codes. *IEEE Transactions on Emerging Topics in Computing*, 9(2), pp. 651–663.

DOI: [10.1109/tetc.2019.2953139](https://doi.org/10.1109/tetc.2019.2953139)

© 2021, IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

Protecting Memories against Soft Errors: The Case for Customizable Error Correction Codes

Jiaqiang Li¹, Pedro Reviriego², Liyi Xiao¹ and Haotian Wu¹

Abstract—As technology scales, radiation induced soft errors create more complex error patterns in memories with a single particle corrupting several bits. This poses a challenge to the Error Correction Codes (ECCs) traditionally used to protect memories that can correct only single bit errors. During the last decade, a number of codes have been developed to correct the emerging error patterns, focusing initially on double adjacent errors and later on three bit bursts errors. However, as the memory cells get smaller and smaller, the error patterns created by radiation will continue to change and thus new codes will be needed. In addition, the memory layout and the technology used may also make some patterns more likely than others. For example, in some memories, there maybe elements that separate blocks of bits in a word, making errors that affect two blocks less likely. Finally, for a given memory, depending on the data stored, some error patterns may be more critical than others. For example, if numbers are stored in the memory, in most cases, errors on the more significant bits have a larger impact. Therefore, for a given memory and application, to achieve optimal protection, we would like to have a code that corrects a given set of patterns. This is not possible today as there is a limited number of code choices available in terms of correctable error patterns and word lengths. However, most of the codes used to protect memories are linear block codes that have a regular structure and which design can be automated. In this paper, we propose the automation of error correction code design for memory protection. To that end, we introduce a software tool that given a word length and the error patterns that need to be corrected, produces a linear block code described by its parity check matrix and also the bit placement. The benefits of this automated design approach are illustrated with several case studies. Finally, the tool is made available so that designers can easily produce custom error correction codes for their specific needs.

Index Terms—Error Correction Codes, Memories, Soft Errors.

1 INTRODUCTION

The wide spread use of electronic circuits, in some cases in critical applications, puts stringent requirements on their reliability. Achieving those can be challenging as electronic devices can suffer a number of failures and errors [1]. For example, defects occur during the manufacturing process that can result in a malfunctioning device and ageing can also lead to failures for devices operating in the field. Another concern are radiation induced soft errors that change the

value of a circuit node and can lead to data corruption [2]. Soft errors are a major concern for memory reliability. To avoid data corruption, memories are commonly protected with Error Correction Codes (ECCs) [3]. Traditionally, the codes used have been able to correct single errors [4] and in some cases also detect double errors [5] as most soft errors affected only a single bit. However, as technology scaled, the probability that a soft error affected more than one bit grew significantly [6] making single error correction insufficient. The bit errors created by a soft error, affect memory cells that are in the area affected by the radiation effect and thus nearby [7]. Therefore, to correct those errors, one option is to use an interleaving of the bits of a word across distant cells so that a soft error corrupts at most one bit per word and thus single error correction is sufficient. Another option is to develop error correction codes that can correct the error patterns created by the soft errors [8]. The error patterns have kept evolving over the last decade [9]. For example, the introduction of FinFET based memories has initially reduced both the soft error rate and the error multiplicity [10]. However, it seems that scaling of FinFETs will not provide further reductions [11]. The error patterns observed in a memory are highly dependent on a number of factors such as the technology used to implement it, the layout and cell structure, the biasing of the memory [11] and the radiation it is exposed to. This has led to the development of a large number of codes that can correct specific patterns. For example, Single Error Correction Double Adjacent Error Detection (SEC-DAEC) [8], [12]; 3-bit bursts [13], [14], [15] or more general patterns [16], [17]. Even for a given memory and radiation environment, depending on the application, we may want to correct different error patterns. For example, if the memory is used to store compressed images, errors on some bits may have a much larger impact than on others so that protection should be better for them [18]. In summary, for each memory/application, a designer would have to carefully understand the protection that would be best and then search on the literature if such a code has already been designed for the required word length.

In this scenario and given that error patterns will most likely continue to evolve, it makes sense rather than presenting new codes for each new set of patterns and word lengths, to present and make available as open source an automatic tool to design the codes. This is indeed possible for linear block codes that use syndrome decoding. The inputs to the tool would be the data word length and the

¹Jiaqiang Li, Liyi Xiao and Haotian Wu are with the Microelectronics Center of the Harbin Institute of Technology. email: jiaqiang.li@cern.ch, xiaoly@hit.edu.cn, doubihaotian@gmail.com

²Pedro Reviriego is with Universidad Carlos III de Madrid, Leganés 28911, Madrid, Spain. email: reviriego@it.uc3m.es

Fig. 1. Systematic structure

error patterns. The tool can then construct a parity check matrix such that all the syndromes that correspond to the specified patterns are different (and thus correctable) and output the matrix and the corresponding bit placement.

In this paper, an automatic tool to design linear block codes that can correct an arbitrary set of error patterns is presented. The tool has been made available as open source¹ so that it can be used by designers. The paper starts with some preliminaries on linear block codes and syndrome decoding to provide the background needed. Then the tool is presented describing it both as a black box looking only at the inputs and outputs and also discussing the implementation and search algorithms used.

To illustrate the proposed approach, two case studies are considered on which there is a need to design specific codes. The first case study considers the design of 3-bit burst codes that do not need to detect or correct errors on the parity bits. This occurs when the ECC is used as a hash function to determine the position on which to store a value [19]. The design of optimized codes for this application enables a reduction in the number of parity check bits needed clearly showing the benefits of tailoring the ECC to the specific requirements. The second case study considers the protection of a memory against double adjacent errors only on the upper bits. In this case, codes that can correct single bit errors on all the bits and double adjacent errors in the upper bits while detecting double adjacent bit errors on the remaining bits are designed. The codes require in most cases the same number of parity check bits as a SEC code and thus illustrate how with the same redundancy, additional protection can be implemented by designing specific codes. The proposed tool is able to find codes for each of the case studies making the whole process much more efficient and friendly to the designer. This shows the potential benefits of an automated and customized ECC design for the protection of memories. The paper ends with the conclusions and some ideas to extend the work presented in several directions.

2 LINEAR BLOCK CODES

Linear block codes are a class of error correction codes that operate on blocks of data [20]. This is suitable for memories as they also operate on blocks (words) of data. A linear block code is defined by its generating and parity check matrices and they are commonly used to protect memories [21]. Encoding and syndrome computation can be seen as matrix multiplication operations as described in the following.

For a message block u (k data bits), there are 2^k possible different messages. According to certain rules, the message u can be transferred into an n -dimension vector v , defined as the codeword of the message u . The set of 2^k codewords

1. The code will be made available as open source when the paper is published.

v defines the block code. If there are k linear separate codewords $g_0, g_1, \dots, g_{k-1}$ and any codeword v can be represented as the linear combination of the k separate codewords as shown in (1), the block code is a linear block code.

$$v = u_0 g_0 + u_1 g_1 + \dots + u_{k-1} g_{k-1} \quad (1)$$

where u_i is 0 or 1, ($0 \leq i < k$). Take the k linear separate codewords as the row vectors, $matrix_{k \times n}$ can be obtained in (2):

$$G = \begin{bmatrix} g_0 \\ g_1 \\ \vdots \\ g_{k-1} \end{bmatrix} = \begin{bmatrix} g_{00} & g_{01} & \cdots & g_{0,n-1} \\ g_{10} & g_{11} & \cdots & g_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ g_{k-1,0} & g_{k-1,1} & \cdots & g_{k-1,n-1} \end{bmatrix} \quad (2)$$

where $g_i = (g_{i0}, g_{i1}, \dots, g_{i,n-1})$, $0 \leq i < k$. If the message $u = (u_0, u_1, \dots, u_{k-1})$ is the input data bit, then the corresponding codeword v can be obtained in (3):

$$v = u \times G = (u_0, u_1, \dots, u_{k-1}) \cdot \begin{bmatrix} g_0 \\ g_1 \\ \vdots \\ g_{k-1} \end{bmatrix} \quad (3)$$

Obviously, for a code (n, k) , if a k -bit input data is given, the n -dimension codeword v can be generated by the matrix G , defined as the generator matrix of the linear block code (n, k) . If the codeword v has the structure shown in Figure 1, the linear block code is defined as the systematic block code and its generator matrix has the systematic structure $G = [PI_k]$ is:

$$G = \begin{bmatrix} p_{00} & \cdots & p_{0,n-k-1} & 1 & 0 & \cdots & 0 \\ p_{10} & \cdots & p_{1,n-k-1} & 0 & 1 & \cdots & 0 \\ p_{20} & \cdots & p_{2,n-k-1} & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ p_{k-1,0} & \cdots & p_{k-1,n-k-1} & 0 & 0 & \cdots & 1 \end{bmatrix} \quad (4)$$

Then, based on the structure of matrix G , the codeword $v = (v_0, v_1, \dots, v_{n-k-1}, v_{n-k}, \dots, v_{n-1})$ can be obtained in (5):

$$v_i = \begin{cases} u_0 p_{0i} \oplus u_1 p_{1i} \oplus \cdots \oplus u_{k-1} p_{k-1,i} & i \in [0, n-k-1] \\ u_i & i \in [n-k, n-1] \end{cases} \quad (5)$$

For a systematic code, the parity bit can be calculated by the xor value of the relative data bits that are decided by the parameter p_{ij} in the matrix. The logic depth of the encoding process is decided by the number of the case ($p_{ij} = 1$) in a column of the matrix.

In a similar way, for the linear block code, there is another matrix called parity-check matrix (H) that generates its dual code. If v is one of the codewords generated by matrix generator G , v is orthogonal to the transpose of the parity-check matrix H :

$$v \cdot H^T = 0 \quad (6)$$

The systematic structure of the parity-check matrix $H = [I_{n-k} P^T]$ is in (7), the encoding process by using the H matrix also obeys the rules in (5).

$$H = \begin{bmatrix} 1 & 0 & \cdots & 0 & p_{00} & \cdots & p_{k-1,0} \\ 0 & 1 & \cdots & 0 & p_{01} & \cdots & p_{k-1,1} \\ 0 & 0 & \cdots & 0 & p_{02} & \cdots & p_{k-1,2} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 & p_{0,n-k-1} & \cdots & p_{k-1,n-k-1} \end{bmatrix} \quad (7)$$

Fig. 2. Illustration of the encoding and decoding process of a SEC-DAEC code

For a code (n, k) , $v = (v_0, v_1, \dots, v_{n-1})$ is the encoded codeword to be stored in memory, $r = (r_0, r_1, \dots, r_{n-1})$ is the codeword read from the memory and $e = (e_0, e_1, \dots, e_{n-1})$ is the error pattern induced by the radiation soft error. The error pattern e can be obtained as:

$$e = r \oplus v \quad (8)$$

if $e = 0$, no error occurs in the received codeword:

$$r \cdot H^T = (e \oplus v) \cdot H^T = v \cdot H^T = 0 \quad (9)$$

if $e \neq 0$, there are errors in the received codeword:

$$r \cdot H^T = (e \oplus v) \cdot H^T = e \cdot H^T \oplus v \cdot H^T = e \cdot H^T \quad (10)$$

Here, we introduce a variable called syndrome (s) to implement the error detection and correction as shown in (11).

$$s = r \cdot H^T = (s_0, s_1, \dots, s_{n-k-1}) \quad (11)$$

Based on equations (7) and (11), the syndrome can be calculated as:

$$\begin{aligned} s_0 &= r_0 + r_n \cdot k p_{00} + r_{n+k+1} p_{10} + \dots + r_{n+1} p_{k-1,0} \\ s_1 &= r_1 + r_n \cdot k p_{01} + r_{n+k+1} p_{11} + \dots + r_{n+1} p_{k-1,1} \\ &\vdots \\ s_x &= r_x + r_n \cdot k p_{0,x} + r_{n+k+1} p_{1,x} + \dots + r_{n+1} p_{k-1,x} \end{aligned} \quad (12)$$

where $x = n - k - 1$. The logic depth of the syndrome calculation is also decided by the number of the ones, ($p_{ij} =$

1), in a row of the matrix. If $e = 0$, then $s = 0$; If $e \neq 0$, then $s = e \cdot H^T \neq 0$. The syndrome vector s can be obtained as:

$$\begin{aligned} s_0 &= e_0 + e_n \cdot k p_{00} + e_{n+k+1} p_{10} + \dots + e_{n+1} p_{k-1,0} \\ s_1 &= e_1 + e_n \cdot k p_{01} + e_{n+k+1} p_{11} + \dots + e_{n+1} p_{k-1,1} \\ &\vdots \\ s_x &= e_x + e_n \cdot k p_{0,x} + e_{n+k+1} p_{1,x} + \dots + e_{n+1} p_{k-1,x} \end{aligned} \quad (13)$$

where $x = n - k - 1$. The syndrome vector s equals to the xor of the corresponding matrix column vectors in the error pattern. For correction, if one type of error pattern can be corrected, every syndrome of the error pattern should be unique. Through the syndromes comparison, the error pattern can be identified and corrected.

The encoding and decoding of a SEC-DAEC (22, 16) code is illustrated in Fig.2. The upper part of the figure shows the computation of the parity bits (encoding) and the lower part the decoding process that starts with the syndrome computation. Then, the position of the incorrect data bits can be located by comparing the syndrome vectors with that belonging to the correctable error patterns. Once the error is located, before the data bits are output, we flip the incorrect data bits to get the correct data bits.

From this brief discussion of linear block codes, it can be seen that the error patterns to be corrected or detected dictate the requirements that the parity check matrix has to meet. Therefore, it is possible to define a tool that automates the process of finding such matrix. This is discussed in the following section.

3 AUTOMATED ECC DESIGN TOOL (AEDT)

This section describes the tool developed to automate ECC design for memory protection. The tool is first presented as a black box showing how it can be used to design specific codes. Then some details on its implementation are briefly discussed. The tool is publicly available in ².

3.1 AEDT parameters and usage

The proposed tool takes as input the error patterns and then formulates the conditions that the matrix needs to have and uses a search program to find matrices that meet them. This process is general and only limited by the flexibility of defining the error patterns. To make the tool as general as possible, it supports a flexible definition of the patterns to correct or detect. This is done in an input file that specifies, the patterns to be detected or corrected and the bits for which each pattern applies. The input file also allows to select the bit placement for the parity bits as that can be used to optimize the code in some cases [15]. In more detail, the input file contains the following information:

- Number of data bits.
- Number of parity check bits.
- Error patterns to correct.
- Bit ranges for which each error correction pattern applies.
- Error patterns to detect.
- Bit ranges for which each error detection pattern applies.
- Positions to place the parity bits in the word.

The parameters are summarized in Figure 3 that shows also the parameters of the tool command line that define the number of correctable and detectable error patterns and the amount of time that the tool is allowed to search for the code.

Fig. 3. Input Parameters for the proposed tool

The parameters are better described with an example. Let us consider the design of a code with $k = 16$ data bits, $n - k = 6$ parity check bits that implements SEC-DAEC on data bits and SED-DAED on parity bits. The parity bits are placed at the beginning of the word. The input file for AEDT would be as shown in Figure 4. It can be seen that the error patterns are defined as an array with '1' on the positions in error and '0' on the ones with no error. For example, pattern '101' would correspond to a double almost adjacent error. In

this example, only single '1' and double adjacent '11' error patterns are used. They are corrected on data bits (bits 7-22) and detected on parity bits (1-6). From the example, it can be seen that this format supports a flexible definition of the error detection/correction capabilities of the ECC and will support many specific designs as will be seen in the next section that discusses two case studies.

```

*****
* Data bits
16
*****
* Parity bits
6
*****
* Error patterns to correct
1 0 0 0 0 0 0 0 0
1 1 0 0 0 0 0 0 0
*****
* Bits to apply each pattern
7 22 0 0 0 0 0 0 0
7 22 0 0 0 0 0 0 0
*****
* Error patterns to detect
1 0 0 0 0 0 0 0 0
1 1 0 0 0 0 0 0 0
*****
* Bits to apply each pattern
1 6 0 0 0 0 0 0 0
1 6 0 0 0 0 0 0 0
*****
* Parity bit placement
1 2 3 4 5 6
  
```

Fig. 4. Input file to the tool to design a code with SEC-DAEC on data bits and SED-DAED on parity bits

The last parameter needed to run the tool is the time allowed to search for the code. This is given in seconds in the command line as a parameter to the executable. The tool will provide the best matrix found (if any) on the time given. This enables the designer to control the execution time. A more detailed explanation of the parameters and instructions on how to use the tool are given in the Readme file provided with the source code.

3.2 Search Algorithm Description

The algorithm used to design the codes solves the Boolean satisfiability problem of finding a matrix that meets the desired constraints and it is a generalized version of the programs used in some of our previous works, for example [15]. The program starts by adding columns to the matrix and checking on each step that the partial matrix meets the syndrome requirements. If not, columns are removed and then other columns are added using backtracking. The checking is done based on the input patterns selected by the user as discussed in the previous subsection. The user can also decide the placement of the parity bits and the search algorithm would then force those positions to store a parity bit. Finally, to minimize the number of ones in the matrix and in the row with more ones, preference is given to columns with fewer ones when trying to add new columns

2. A link to github will be provided when the paper is published

Fig. 5. Block diagram of the algorithm implemented in the AEDT

to the matrix during the construction as long as a solution is found.

The block diagram of the algorithm is shown in Fig.5. The parameters of the input file are read and used to compute the error patterns that need to be corrected or detected. For each of them, a function is run to obtain the corresponding syndromes that need to be checked and it is applied only for the range specified in the input parameters. All those functions are combined in a general function that gives for each position the syndromes that need to be checked assuming all previous columns have been already added to the matrix. Then, when adding a column we first check that the checking function does not return a conflict with any of the existing syndromes from previous columns (Syndrome set) and if so we add the new column to the current matrix (Base matrix) and the syndromes for the new column to the set of syndromes already used. The use of a generic format for the input file parameters and for the checking functions ensure that the algorithm is flexible enough to cover many practical cases.

To optimize the designed codes, the tool tries to min-

imize the number of ones as that reduces the circuitry for encoding and syndrome checking. To do so, the first candidate columns to be added to the matrix are those with fewer ones. Only when using those does not lead to a solution, columns with larger weight are used.

Finally, as the user can select the placement of the parity bits, when selecting the candidate column, we first check if it is a parity bit or a data bit and then select only among parity columns or data columns respectively. At the end of each iteration, the time is checked and when the tool has run for the duration selected by the user, the program stops regardless of whether a matrix has been found.

4 CASE STUDIES

To illustrate the usage of the proposed tool and its benefits, two case studies are presented in this section. The first one is the use of ECCs in hash tables as proposed in [19]. The idea is to use ECCs as the hash functions of the tables so that they can also provide error detection and correction. In this application, the parity bits are the address on which the data

is stored and are not subject to errors. Therefore, the ECCs can be optimized as they do not need to detect or correct errors on the parity bits. The second case study considers additional protection for the upper bits in a memory word. This is useful in some applications on which errors on the most significant bits have a larger impact [18]. In this second case study, an automated tool gives the designer the freedom to select the number of upper bits to protect so that the code matches the application requirements.

Hash Code:

* Data bits
12

* Parity bits
6

* Error patterns to correct
1 0 0 0 0 0 0 0 0 0
1 1 0 0 0 0 0 0 0 0
1 0 1 0 0 0 0 0 0 0
1 1 1 0 0 0 0 0 0 0

* Bits to apply each pattern
7 18 0 0 0 0 0 0 0 0
7 18 0 0 0 0 0 0 0 0
7 18 0 0 0 0 0 0 0 0
7 18 0 0 0 0 0 0 0 0

* Parity bit placement
1 2 3 4 5 6

Fig. 6. Input to the tool for the hash code

4.1 Hash ECC

In many cases, memories are used to store hash tables on which an given element x is mapped to one or more positions using hash functions $h(x)$ and stored in one of them. One possible optimization is to use an ECC as the hash function as discussed in [19]. In this configuration, errors can be detected and corrected using the address as the parity check bits. This also means that errors will not affect the parity bits as they are not physically stored in the memory and thus the ECC does not need to detect correct errors on those bits. This can potentially be used to optimize ECCs that are used as hash functions.

To illustrate the potential benefits of the proposed AEDT tool, a 3-bit burst code is considered with word lengths of $k = 12, 16, 20, 24$ to be used as a hash function. Since errors in the parity bits cannot occur, the code can be optimized to correct errors only on the data bits. This can reduce the number of parity check bits as the number of syndromes needed are significantly reduced. The number of ones in the matrix can also be reduced and that should result in more efficient encoding and syndrome checking. For example, for $k = 12$ the input to the tool is shown in Fig.6.

The codes obtained are given in Figures 7,8,9,10. It can be observed that some columns with weight one are

duplicated. This is possible because there are no errors on parity bits and thus weight one columns can be used for data columns. The main parameters of the code, namely the number of parity check bits, the total number of ones in the matrix and the maximum number of ones in a row are summarized in Table 1 and compared to that of general 3-bit burst codes. It can be seen that the optimized codes reduce the number of parity bits by one (thus lowering the memory overhead) when $k = 12, 24$ and reduce the number of ones in the matrix and in the heaviest row in most cases. This would result in more efficient encoders and decoders as shown in the evaluation results presented in the next section.

$$H = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

Fig. 7. Hash ECC 3-bit burst code for $k = 12$

$$H = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Fig. 8. Hash ECC 3-bit burst code for $k = 16$

$$H = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

Fig. 9. Hash ECC 3-bit burst code for $k = 20$

$$H = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

Fig. 10. Hash ECC 3-bit burst code for $k = 24$

4.2 Double Adjacent Error Correction on Upper Bits

There are applications like signal processing on which the impact of an error is quite different depending on its position. In general, errors on the upper or more significant bits tend to have a large impact while errors on the lower bits may even be acceptable. This has motivated the design of unequal error correction codes that provide additional protection for the upper bits [18]. In this case, to obtain better results, it would be ideal to be able to tailor the ECC

Data Bits	Traditional 3-Bit Burst Code				Optimized 3-Bit Burst Code			
	Parity Bits	Syndromes Needed	Total Ones	Ones in Heaviest Row	Parity Bits	Syndromes Needed	Total Ones	Ones in Heaviest Row
12	7	71	35	5	6	43	25	5
16	7	87	46	8	7	59	33	5
20	7	103	57	9	7	75	44	7
24	8	123	66	10	7	91	54	8

TABLE 1: Hash ECC versus Traditional 3-Bit Burst Code Comparison.

Data Bits	SEC Code				DAEC on Upper Bits			
	Parity Bits	Syndromes Needed	Total Ones	Ones in Heaviest Row	Parity Bits	Syndromes Needed	Total Ones	Ones in Heaviest Row
8	4	12	22	6	5	18	23	5
12	5	17	31	7	5	23	35	7
16	5	21	43	9	5	28	49	11
24	5	29	71	15	6	40	68	12
32	6	38	87	15	6	50	95	16

TABLE 2: Comparison of SEC versus SEC with DAEC on upper bits

to the application requirements in terms of the number of bits that need additional protection. Again, a tool that can design this custom ECCs would be useful.

To illustrate the benefits of customizing the ECCs, a SEC code that is extended to provide also double adjacent error correction for some bits is used. In particular, data word lengths of $k = 8, 12, 16, 24, 32$ with double adjacent error protection on the upper 2, 3, 4, 6, 8 bits respectively are considered as shown in Fig.11. As an example, the input to the tool for $k = 8$ is shown in Fig.12.

Fig. 11. Cases considered for unequal protection of upper bits

The codes obtained are given in Figures 13 to 17 and compared to existing SEC codes in Table 2. It can be observed that the proposed codes require an additional parity bit only when $k = 8, 24$ and thus in many cases no memory overhead is needed compared to SEC codes. Instead, the number of ones is larger in most cases which would result in a more complex circuitry. This is the price paid for the additional protection of the upper bits.

In this second case study, since the codes needed are not readily available, without the ADET tool, the designer would need to write a program to find the matrices or else use a SEC-DAEC code that covers all bits which would require an additional parity bit. Therefore, the benefits of an automated tool can again be clearly seen.

5 EVALUATION

In order to evaluate the benefits of the codes generated by the proposed tool, the encoders and decoders have been

DAEC on Upper Bits Code:	

* Data bits	8

* Parity bits	5

* Error patterns to correct	1 0 0 0 0 0 0 0 0
	1 1 0 0 0 0 0 0 0

* Bits to apply each pattern	1 13 0 0 0 0 0 0 0
	12 13 0 0 0 0 0 0 0

* Error patterns to detect	1 1 0 0 0 0 0 0 0

* Bits to apply each pattern	1 12 0 0 0 0 0 0 0

* Parity bit placement	1 2 3 4 5

Fig. 12. Input to the tool for the DAEC on upper bits code

$$\mathbf{H} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Fig. 13. SEC code with DAEC on upper bits for $k = 8$

implemented in HDL and mapped to a TSMC 65nm library using Synopsis Design Compiler. The alternative traditional codes have been also implemented for comparison. The synthesis was done with two configurations, maximum effort on area optimization and maximum effort on delay optimization. These settings should give an indication of the lowest area and delay that can be achieved respectively.

The synthesis results are presented on Tables 3 and 4 for the hash 3-bit burst codes and upper bit DAEC codes case studies. Table 3 compares the optimized 3-bit burst

Data Bits	Code	Parity Bits	Area Optimization						Delay Optimization					
			Encoder			Decoder			Encoder			Decoder		
			Area	Delay	Power	Area	Delay	Power	Area	Delay	Power	Area	Delay	Power
12	Tradition	7	123.60	1.08	0.08	363.20	3.76	0.57	279.99	0.28	0.13	2340.39	0.66	2.11
	Optimized	6	95.60	1.06	0.05	244.80	3.23	0.33	219.59	0.26	0.09	1650.79	0.58	1.31
16	Tradition	7	165.60	1.57	0.11	494.80	4.14	0.87	393.19	0.34	0.19	3413.59	0.71	3.40
	Optimized	7	129.20	1.10	0.07	350.40	3.57	0.56	279.59	0.27	0.12	1807.19	0.64	1.63
20	Tradition	7	239.60	1.62	0.17	631.60	4.76	1.22	547.19	0.35	0.28	3732.39	0.75	4.14
	Optimized	7	193.60	1.10	0.11	488.80	4.93	0.88	467.59	0.32	0.21	3265.99	0.67	3.31
24	Tradition	8	266.80	1.62	0.18	767.60	4.60	1.59	534.39	0.38	0.29	4921.99	0.75	5.44
	Optimized	7	248.80	1.26	0.16	605.19	4.46	1.20	510.79	0.35	0.26	3759.99	0.70	4.34

TABLE 3: Hash ECC overhead comparison. Area expressed in μm^2 , delay in nanosecond and power in milliwatts

Data Bits	Code	Parity Bits	Area Optimization						Delay Optimization					
			Encoder			Decoder			Encoder			Decoder		
			Area	Delay	Power	Area	Delay	Power	Area	Delay	Power	Area	Delay	Power
8	SEC	4	84.80	1.08	0.05	139.60	2.36	0.14	171.19	0.32	0.08	613.99	0.51	0.46
	Proposed Code	5	75.20	1.03	0.04	153.20	2.37	0.16	229.59	0.27	0.09	767.99	0.52	0.55
12	SEC	5	127.20	1.10	0.08	205.20	2.74	0.21	273.59	0.32	0.13	1037.19	0.54	0.86
	Proposed Code	5	132.40	1.61	0.09	210.00	2.45	0.26	293.99	0.34	0.15	979.19	0.59	0.75
16	SEC	5	180.80	1.28	0.12	273.60	3.34	0.31	453.59	0.34	0.23	1469.19	0.60	1.40
	Proposed Code	5	178.40	1.62	0.12	282.00	3.07	0.38	439.59	0.38	0.24	1381.59	0.65	1.37
24	SEC	5	275.60	1.80	0.21	392.40	3.36	0.53	662.39	0.41	0.38	1972.79	0.66	1.92
	Proposed Code	6	272.40	1.46	0.19	435.60	3.56	0.61	528.39	0.41	0.31	2003.99	0.71	2.20
32	SEC	6	369.59	1.48	0.27	536.80	3.89	0.71	857.59	0.41	0.48	2307.19	0.68	2.28
	Proposed Code	6	376.00	1.71	0.28	583.60	4.00	0.95	888.39	0.44	0.52	2792.79	0.73	3.27

TABLE 4: Upper bits DAEC overhead comparison. Area expressed in μm^2 , delay in nanosecond and power in milliwatts

$$\mathbf{H} = \begin{bmatrix} 10000000110011101 \\ 01000101001101100 \\ 00100010001110011 \\ 00010100010110110 \\ 00001011101001010 \end{bmatrix}$$

Fig. 14. SEC code with DAEC on upper bits for $k = 12$

$$\mathbf{H} = \begin{bmatrix} 100000000000111100000001010111 \\ 010000000111000001001110101010 \\ 001000101000000101110000101101 \\ 000100010001001010110110011001 \\ 00001010001001001010101110100 \\ 000001011100100011011101000010 \end{bmatrix}$$

Fig. 16. SEC code with DAEC on upper bits for $k = 24$

$$\mathbf{H} = \begin{bmatrix} 100000000111101111011 \\ 010001101000010011010 \\ 001000011011011100110 \\ 000101001010100110101 \\ 000010110110011001101 \end{bmatrix}$$

Fig. 15. SEC code with DAEC on upper bits for $k = 16$

$$\mathbf{H} = \begin{bmatrix} 1000000000001111000000010111110110101 \\ 01000000011100000100111010011101011010 \\ 0010001010000001011000011100101101110 \\ 00010001000100101011011010001011100101 \\ 00001010001001001010101100110010101010 \\ 00000101110010001101110101000001010011 \end{bmatrix}$$

Fig. 17. SEC code with DAEC on upper bits for $k = 32$

codes that do not detect or correct errors on parity bits with traditional 3-bit burst codes. It can be observed that the proposed codes achieve significant reductions in area, power and delay in most cases. For example, for area optimization the decoder area is reduced by at least 20%. The savings for the encoders are lower but always above 5%. As the area of decoders is much larger than that of encoders, the overall savings would be significant. Looking into the delay optimization and focusing on the decoders that are again the ones with larger values, it can be seen that reductions of at least 9% are obtained. These benefits in encoding and decoding circuitry are in addition to the savings of parity bits discussed in the previous section. In summary, the proposed tool has enabled a reduction in both memory size and encoder and decoder complexity for this first case study.

The results for the DAEC on upper bits are shown in Table 4. In this case, the proposed codes are compared to SEC only codes. Therefore, the proposed codes provide a stronger error protection. From the results, it can be seen that implementing DAEC on upper bits introduces an overhead mostly for the decoder. For area optimization, the penalty in decoder area is below 11% and for delay

optimization, the penalty in delay is below 9%. This shows that the additional protection on the upper bits can be achieved with a reasonable impact on decoding complexity. Instead, going for a full DAEC on all bits would require a larger overhead and in some cases additional parity bits. This second case study shows how the tool can provide codes targeted to the needs of specific applications.

6 CONCLUSION AND FUTURE WORK

In this paper, AEDT, a flexible tool to automate the design of linear block codes for memory protection has been presented. The tool supports a generic definition of the error patterns to detect and correct as well as the ranges of bits to which each pattern applies. It also lets the user select the placement of the parity bits and the execution time. This flexibility enables the design of codes for most current error patterns and also for those expected in the foreseeable future.

To illustrate the applicability and benefits of the proposed tool, two case studies have been considered: the design of codes that are used as hash functions and of

codes with additional protection in a few upper bits. In the first case, the tool enables the designer to easily obtain code matrices that reduce the memory overheads and the encoding and decoding complexity compared to existing codes not tailored to be used as hash functions. In the second case, the designer can protect a few additional bits with limited cost instead of having to use a traditional SEC code.

The proposed tool has been made publicly available so that designers can benefit from it and also researchers can extend or improve the tool. In particular, an interesting addition to the tool would be the generation of HDL code for the encoders and decoders so that the designers do not only get the matrix but also a working implementation. This indeed seems possible by using syndrome decoding. This extension is left for future work.

ACKNOWLEDGMENTS

Jiaqiang Li and Liyi Xiao would like to acknowledge the support of the Fundamental Research Funds for the Central Universities (Grant No.HIT.KISTP.201404), Harbin science and innovation research special fund (2015RAXXJ003), and Special found for development of Shenzhen strategic emerging industries (JCYJ20150625142543456).

Pedro Reviriego would like to acknowledge the support of the TEXEO project TEC2016-80339-R funded by the Spanish Ministry of Economy and Competitivity and of the Madrid Community research project TAPIR-CM grant no. P2018/TCS-4496.

REFERENCES

- [1] N. Kanekawa, E. H. Ibe, T. Suga Y. and Uematsu, "Dependability in Electronic Systems," Springer, New York, 2011.
- [2] R. Baumann, "Soft errors in advanced computer systems," in *IEEE Design Test of Computers*, vol. 22, no. 3, pp. 258-266, May-June 2005.
- [3] C. L. Chen and M. Y. Hsiao, "Error-correcting codes for semiconductor memory applications: a state-of-the-art review," *IBM J. Res. Develop.*, vol. 28, no. 2, pp. 124-134, Mar. 1984.
- [4] R. W. Hamming, "Error detecting and error correcting codes," *The Bell Syst. Tech. J.*, vol. 26, no. 2, pp. 147-160, 1950.
- [5] M. Y. Hsiao, "A class of optimal minimum odd-weight-column SEC-DED codes," *IBM J. Res. Develop.*, vol. 14, no. 4, pp. 395-401, Jul. 1970.
- [6] E. Ibe, H. Taniguchi, Y. Yahagi, K. Shimbo and T. Toba, "Impact of Scaling on Neutron-Induced Soft Error in SRAMs From a 250 nm to a 22 nm Design Rule," in *IEEE Transactions on Electron Devices*, vol. 57, no. 7, pp. 1527-1538, July 2010.
- [7] S. Satoh, Y. Tosaka, and S. A. Wender, "Geometric effect of multiple-bit soft errors induced by cosmic ray neutrons on DRAMs," *IEEE Electron Device Lett.*, vol. 21, no. 6, pp. 310312, Jun. 2000.
- [8] A. Dutta and N. A. Touba, "Multiple bit upset tolerant memory using a selective cycle avoidance based SEC-DED-DAEC code," in *Proc. IEEE VLSI Test Symp.*, May 2007, pp. 349-354.
- [9] L. W. Massengill, B. L. Bhuva, W. T. Holman, M. L. Alles and T. D. Loveless, "Technology scaling and soft error reliability," 2012 IEEE International Reliability Physics Symposium (IRPS), Anaheim, CA, 2012, pp. 3C.1.1-3C.1.7.
- [10] Y. Fang and A. S. Oates, "Characterization of Single Bit and Multiple Cell Soft Error Events in Planar and FinFET SRAMs," in *IEEE Transactions on Device and Materials Reliability*, vol. 16, no. 2, pp. 132-137, June 2016.
- [11] B. Narasimham, S. Gupta, D. Reed, J. K. Wang, N. Hendrickson and H. Taufique, "Scaling trends and bias dependence of the soft error rate of 16 nm and 7 nm FinFET SRAMs," 2018 IEEE International Reliability Physics Symposium (IRPS), Burlingame, CA, 2018, pp. 4C.1-1-4C.1-4.
- [12] Z. Ming, X. L. Yi, and L. H. Wei, "New SEC-DED-DAEC codes for multiple bit upsets mitigation in memory," in *Proc. IEEE/IFIP 20th Int. Conf. VLSI Syst.-Chip*, Oct. 2011, pp. 254-259.
- [13] A. Klockmann, G. Georgakos and M. Goessel, "Systematic Design of a New 3-Bit-Burst-Error Correction Code with Minimal Number of Check Bits," *Reliability by Design*, 9. ITG/GMM/GI-Symposium, Cottbus, Germany, 2017.
- [14] L. J. Saiz-Adalid, P. Reviriego, P. Gil, S. Pontarelli, and J. A. Maestro, "MCU tolerance in SRAMs through low-redundancy triple adjacent error correction," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 23, no. 10, pp. 2332-2336, Oct. 2015.
- [15] J.Q Li, P. Reviriego, L.Y. Xiao, Z. Liu, L. Li and Anees Ullah, "Low delay Single Error Correction and Double Adjacent Error Correction (SEC-DAEC) codes," *Microelectronics Reliability*, vol. 97, pp 31-37, 2019.
- [16] D. Abhishek and N. A. Touba, "Low Complexity Burst Error Correcting Codes to Correct MBUs in SRAMs," in *Proc. ACM Great Lakes Symposium on VLSI*, May 2018, pp. 219-224.
- [17] A. Neale, M. Jonkman and M. Sachdev, "Adjacent-MBU-Tolerant SEC-DED-TAEC-yAED Codes for Embedded SRAMs," in *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 62, no. 4, pp. 387-391, April 2015.
- [18] X. Yang and K. Mohanram, "Unequal-error-protection codes in SRAMs for mobile multimedia applications," in *Proc. IEEE/ACM Int. Conf. Comput.Aided Design*, 710, Nov. 2011, pp. 2127.
- [19] P. Reviriego, S. Pontarelli, J.A. Maestro, M. Ottavi, "Efficient Implementation of Error Correction Codes in Hash Table," *Microelectronics Reliability*, vol. 54, no. 1, January 2014, pp. 338-340.
- [20] S. Lin and D. J. Costello, "Error Control Coding," 2nd ed. Englewood Cliffs, NJ, USA: Prentice-Hall, 2004.
- [21] E. Fujiwara, "Code Design for Dependable Systems," 1st ed. Hoboken, NJ, USA: Wiley, 2006, ch. 2, sec. 2, pp. 2376.