Design and Analysis of Efficient Synthesis Algorithms for EDAC Functions in FPGAs

Carlos Colodro-Conde, Rafael Toledo-Moreo

***Abstract*—Error Detection and Correction (EDAC) functions have been widely used for protecting memories from single event upsets (SEU), which occur in environments with high levels of radiation or in deep submicron manufacturing technologies. This paper presents three novel synthesis algorithms that obtain area-efficient implementations for a given EDAC function, with the ultimate aim of reducing the number of sensitive configuration bits in SRAM-based Field-Programmable Gate Arrays (FPGAs). Having less sensitive bits results in a lower chance of suffering a SEU in the EDAC circuitry, thus improving the overall reliability of the whole system. Besides minimizing area, the proposed algorithms also focus on improving other figures of merit like circuit speed and power consumption. The executed benchmarks show that, when compared to other modern synthesis tools, the proposed algorithms can reduce the number of utilized look-up tables (LUTs) up to a 34.48%. Such large reductions in area usage ultimately result in reliability improvements over 10% for the implemented EDAC cores, measured as MTBF (Mean Time Between Failures). On the other hand, maximum path delays and power consumptions can be reduced up to a 17.72% and 34.37% respectively on the placed and routed designs.**

***Index Terms*—field programmable gate arrays (FPGA), synthesis tools, efficient synthesis, error detection and correction (EDAC), single event upsets (SEU)**

I. Introduction

With the continuous downscaling of the VLSI fabrication technologies, radiation induced errors have become a major concern in modern digital electronics. Even at ground level, high-energy particles like neutrons coming from the cosmic background create undesired current pulses that may invert the value stored in a memory element such as a flip-flop [8], [18], [32]. This kind of errors, called single event upsets (SEU), compromise the reliability of the systems if no action is taken to mitigate them. In the space environment, outside the protection of the magnetosphere of the Earth, SEUs become a critical concern because of the high radiation levels. SEUs can have serious consequences for the spacecraft, including loss of information, functional failure or loss of control [5].

Due to their flexibility and high computing capability, Field-Programmable Gate Arrays (FPGAs) are nowadays widely used to implement digital systems. In aerospace applications, antifuse FPGAs have been traditionally preferred due to their high tolerance to radiation [39]. Nevertheless, there is a growing interest in using SRAM-based FPGAs in space embedded systems because of their lower costs, higher performances and in-flight reconfigurability [28], [36], [43].

C. Colodro-Conde and R. Toledo-Moreo are with the Department of Electronics and Computer Technology, Universidad Politécnica de Cartagena (Spain). Emails: carlos.colodro@upct.es, rafael.toledo@upct.es

SRAM-based FPGAs are specially sensitive to radiation because of the fact that the configuration memory that defines the circuitry implemented inside them is vulnerable to SEUs. Regarding this matter, Hamming EDAC (Error Detection And Correction) cores with SEC-DED (Single-Error-Correction, Double-Error-Detection) capabilities have proven to be an effective way to protect the internal configuration memory [1], [17], [25], [29], [30], as well as other external volatile memories [6], [23], [24]. However, the fact of such cores being placed inside the FPGA makes them less reliable, as they can also be affected by SEUs in configuration memory.

The aim of this work is to develop new synthesis algorithms capable of minimizing the area utilization of SEC-DED EDAC functions in FPGAs without affecting negatively other parameters like processing speed and power consumption. A reduction in area means a lower number of sensitive configuration bits (i.e., bits which are associated with the circuitry of the design), and therefore more reliability of the EDAC cores themselves when they are implemented in SRAM-based FPGAs. This statement is justified by the fact that, under the same radiation environment (modelled by the particle flux) and FPGA model (which determines the area cross-section per bit), the *Mean Time Between Failures* (MTBF) is directly proportional to the number of sensitive bits of the design [22], [41], [42]. The EDAC cores can also get an increase in reliability when implemented in other FPGA technologies (e.g., flash-based), as every instantiated component has an intrinsic rate of failure [26]. On the other hand, having higher processing speeds not only has a positive impact on memory throughput, but also on the reliability of the whole system, as higher scrubbing rates would be possible [7], [38]. Efficient implementations are desirable from an economic perspective as well because they allow choosing smaller FPGAs with lower speed grades.

The algorithms presented in this paper achieve a significant reduction in area, maximum path delay and power consumption figures of the implemented circuits (up to 34.48%, 17.72% and 34.37%, respectively) when compared to the ones obtained by means of commercial synthesis tools. The resulting increase of the MTBF (up to 10.91%) gives an extra protection to these critical functions, which can be easily combined with other well-known techniques like TMR for further increasing the immunity to SEU.

The paper is structured as follows. Section II presents the related works. Section III briefly explains the theory behind EDAC codes. Section IV describes the proposed algorithms formally. Section V compares the performance of the developed methods with other modern synthesis tools. Finally, Section VI draws the main conclusions.

II. Related works

Most of the previous work about optimizing the implementation of SEC-DED EDAC has been aimed to ASIC (Application-Specific Integrated Circuit) devices. The optimization goal is usually the number of transistors, although the speed of the resulting circuit is also considered in some cases [4], [23], [24]. In FPGA devices, as opposed to ASICs, one does not have the freedom to create custom cells at layout level, so the optimization cannot be done following this approach. Another difference is that the main parameters that determine area utilization in FPGAs are usually the number of utilized LUTs and flip-flops, rather than the number of transistors. In a 6-input LUT FPGA, the hardware cost of instantiating any logic function up to 6 inputs is virtually the same, while in ASICs the number of inputs of a gate makes a great difference. Because of these reasons, specific optimization methods were developed for this paper, with the focus on FPGA devices.

Other works in the literature do focus on optimizing the implementation of EDAC codes in FPGAs, but not on the kind of EDAC codes that will be discussed in our paper. For example, [3] introduces an alternative implementation of Reed-Solomon (RS) codes [40] with reduced area utilization, in the context of satellite communications. Despite mentioning the importance of achieving high processing speeds and low power consumption in space applications, the authors of that work do not include results about these two parameters.

There is another work [37] whose main aim is to obtain small and fast implementations of RS codes in FPGAs by choosing the most suitable generator polynomials and multiplication constants. In that work, the optimized RS codes are intended to be used in combination with SEC-DED Hamming codes for protecting memories from SEU, but the optimization of the latter part is not considered. The SEC-DEC Hamming codes are just the ones that the present paper will focus its interest on.

In addition to the novelty of optimizing the implementation of SEC-DEC Hamming codes on FPGAs, the present work proposes new custom synthesis algorithms as an alternative way to obtain such optimized implementations. The analyses and in-depth descriptions presented in the paper allow to obtain a comprehensive knowledge of the problem of synthesizing EDAC functions efficiently from a hardware, a mathematical and a computational perspective.

In the literature on the field, the synthesis process for FPGAs has been traditionally divided in two stages: logic optimization [11], [14] and technology mapping [9], [16]. Recent research on this subject has been focused on combining both stages in order to provide better results [10], [35], although according to [15], [31] there is still much room for improvement. The proposed algorithms exploit this latter idea to its fullest, making no distinction between logic optimization and technology mapping stages.

In contrast to most of the synthesis tools and algorithms, which provide high flexibility at the expense of optimality, this paper focuses on a particular type of problem and tries to find the best possible solution for a given optimization criterion, which can be configured as desired. This strategy will allow a significant reduction in area, maximum path delay and power consumption of the implemented circuits when compared to the ones obtained by means of commercial synthesis tools, which represent the state of art of FPGA synthesis algorithms. Additionally, the obtained solutions can be used to measure the quality of faster, heuristic approaches. In particular, this work presents a greedy algorithm whose use is strongly justified in spite of not being completely optimal, as it has viable execution times even for the largest designs.

III. SEC-DED EDAC codes

The present study will focus on the odd-weight column EDAC codes proposed in [26]. These codes are extensively used in many applications as a result of their SEC-DED (Single-Error-Correction, Double Error Detection) capabilities and their relatively low hardware needs [4], [12], [27].

According to coding theory [21], a SEC-DED EDAC code can be defined via its parity-check matrix $\mathbf{H}_{r\times n} = \{h_{ij}\}$. This matrix has r rows and $n = r+k$ columns, being r the number of parity bits, k the number of data bits and n the codeword length. The codeword $\mathbf{m} = \{m_i\}$ is formed by concatenating the input data bits $\mathbf{d} = \{d_i\}$ and the calculated check bits $\mathbf{c} = \{c_i\}$, and it is the one that is actually stored in memory for protecting the original data.

For calculating the bit i ($1 \leq i \leq r$) of the check bits $\mathbf{c}$, one has to take the row i from the $\mathbf{H}$ matrix and check the positions where $h_{ij} = 1$, with $1 \leq j \leq k$. These positions indicate the elements of the data bits vector $\mathbf{d}$ that have to be XOR'ed between themselves in order to obtain c_i.

In most applications, the codeword $\mathbf{m}$ is saved in a memory which may suffer from undesired bit alterations caused by SEU. When the data needs to be recovered, the syndrome vector $\mathbf{s} = \{s_i\}$ has to be calculated in order to know if the codeword has been altered. With SEC-DEC codes, the syndrome vector can be used to spot the location of single-bit errors so that they can be corrected with a bit flip. If the obtained syndrome is equal to $\mathbf{0}$, it means that the retrieved codeword is the same as the one that was originally saved. If the syndrome is not equal to $\mathbf{0}$ and the parity of the syndrome is even, a double-bit error flag can be raised.

For calculating the bit i ($1 \leq i \leq r$) of the syndrome vector $\mathbf{s}$, one has to take the row i from the $\mathbf{H}$ matrix and check the positions where $h_{ij} = 1$, with $1 \leq j \leq n$. These positions indicate the elements of the codeword $\mathbf{m}$ which have to be XOR'ed between themselves in order to obtain s_i.

Let us illustrate the procedure explained above with the following example $\mathbf{H}$ matrix:

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

The $\mathbf{H}$ matrix above represents the only odd-weight-column SEC-DEC code that generates an 8-bit length codeword ($n = 8$) for a 4-bit data word ($k = 4$). In coding theory, this is

denoted as an (8,4) code. For $\mathbf{H}$ under consideration, the check bits shall be calculated as follows:

$$\begin{aligned} c_1 &= d_1 \oplus d_2 \oplus d_3 \\ c_2 &= d_1 \oplus d_2 \oplus d_4 \\ c_3 &= d_1 \oplus d_3 \oplus d_4 \\ c_4 &= d_2 \oplus d_3 \oplus d_4 \end{aligned} \tag{2}$$

Assuming an input data vector $\mathbf{d} = [1011]$ and applying (2), the check bits would be $\mathbf{c} = [0010]$. The codeword, that is, the actual bits that will be saved in memory would be $\mathbf{m} = [10110010]$.

According to the procedure described previously, the syndrome shall be calculated with the following equations:

$$\begin{aligned} s_1 &= m_1 \oplus m_2 \oplus m_3 \oplus m_5 \\ s_2 &= m_1 \oplus m_2 \oplus m_4 \oplus m_6 \\ s_3 &= m_1 \oplus m_3 \oplus m_4 \oplus m_7 \\ s_4 &= m_2 \oplus m_3 \oplus m_4 \oplus m_8 \end{aligned} \tag{3}$$

If neither the data bits or the check bits have been altered, the resulting syndrome is $\mathbf{s} = [0000]$, as expected. However, if we flip m_3, for example, we would get $\mathbf{s} = [1011]$. By inspecting $\mathbf{H}$, we can spot that the error occurred at d_3, as $\mathbf{s} = [1011]$ matches with the third column of $\mathbf{H}$.

IV. Description of the algorithms

The purpose of the proposed algorithms is to synthesize a given EDAC function in such a way that it allows an efficient implementation on the desired FPGA technology. The optimization goals followed in this study are listed and justified in Section IV-A.

The algorithm presented in Section IV-C guarantees optimality because the whole space of solutions is explored, avoiding to make assumptions based on heuristics. This approach is often categorized as backtracking, and it has the drawback of having long processing times. In order to reduce the cost of blind backtracking, a dynamic programming version of the same algorithm will be presented in IV-D, and its processing speed will be measured later in V-B. In spite of being implemented differently, both versions of the algorithm are functionally equivalent, and they will be referred to with the same name: *FS-EDAC* (Full Search EDAC).

The results provided by any of the versions of FS-EDAC are useful to set an optimal reference that can be used measure the performance of the available synthesis tools, and to develop faster algorithms that try to reach the best possible solution. Following this idea, Section IV-E presents *G-EDAC* (Greedy EDAC): a greedy variant of FS-EDAC whose performance will be evaluated later in Section V, both in terms of processing speed and hardware implementation results.

A. Optimization goals

The algorithms will focus their effort in reducing area occupation, although some attention will also be put in optimizing processing speed and power consumption.

The area occupation can be calculated as the number of utilized LUTs, which are the basic combinational resources available in every FPGA technology. Other specialized FPGA resources such as carry logic or embedded multipliers will be ignored as they cannot be exploited for implementing the circuits under study. For measuring processing speed, path delays will be considered instead of clock periods, as the inferred circuits will be purely combinational. Optionally, higher speeds can be achieved by inserting flip-flops within the internal stages to form a pipeline, but at the cost of more area, specially if Triple Modular Redundancy (TMR) is used to protect such flip-flops against radiation.

The actual maximum path delay, which will determine the processing speed, can only be obtained by the timing analysis tools provided by the selected FPGA vendor after the synthesis, place and route stages. Because of this reason, the parameter to minimize will not be the maximum path delay itself. Instead, other variables which are directly related to the maximum path delay will be considered, namely:

- The maximum number of LUTs that a path has to go through (from now on, *levels*). Each LUT adds a certain delay to the path, as well as the nets used to connect two consecutive LUTs.
- The number of nets or signals that are part of the circuit. The higher the number of nets, the more complex will get the routing process, which generally results in longer delays.
- The maximum fan-out of all the LUTs present in the circuit. A higher fan-out in an output means that it takes more time for this output to charge the input capacitances of the inputs it is connected to, as the current is divided among those inputs.

Both the number of utilized LUTs and the number of nets have an impact on power consumption. As these parameters will be minimized by the algorithms, a reduction on power consumption can also be expected for a given clock frequency.

Given that not all of the mentioned parameters can be minimized at the same time, some type of priority mechanism has to be established. The proposed algorithms prioritize the minimization of the parameters with the following order (highest priority first):

1) Number of LUTs,
2) Maximum level.
3) Number of nets.
4) Maximum fan-out.

In all cases, the optimization consists on minimizing the value of the parameter.

B. Algorithm interface

The inputs to all the algorithms presented in this section are $\mathbf{H}_{r\times n} = \{h_{ij}\}$, $l(j)$ and K, while the output is the optimized netlist. Each one of the inputs will be described in the paragraphs below.

The parity-check matrix $\mathbf{H}$ defines the EDAC function to implement. It consists of a binary matrix ($h_{ij} = 0$ or 1) where each of the r rows represents an output, and each of the n columns represents an input. As explained in Section III, an output i is calculated as the XOR combination of all the inputs of the corresponding row that satisfy the condition $h_{ij} = 1$.

The function $l(j)$ establishes the level of the jth input ($l(j) \geq 0$). It is common that the level of every input signal comes directly from the output of a flip-flop. For that particular case, we would have $l(j) = 0$ for $1 \leq j \leq n$.

The constant K is the maximum number of inputs that a LUT of the target technology has, so the algorithms may instantiate LUTs with a number of inputs between 2 and K ($K \geq 2$). Other characteristics of the selected FPGA technology like the routing architecture are not taken into account.

C. Backtracking version

This section describes the backtracking version of FS-EDAC. The steps of the algorithm are summarized in the list below, and they will be explained extensively throughout the subsequent subsections:

1) Initialization
2) Search for common terms
3) Update of the current network
4) Parameter calculation
5) Solution evaluation
6) Construction of the final network

Steps 3 to 5 will be applied to each *common term* found in Step 2. A common term is defined as a group of two or more terms in one output (i.e., a column of $\mathbf{H}$) that are shared at least with another output. At the same time, Steps 2 to 5 will be applied to each network obtained in Step 3. This means that FS-EDAC is a highly recursive algorithm, as it enters a new level of recursion with each found common term, including the ones obtained as a result of having grouped previous common terms. Therefore, a long execution time can be expected when the input functions have a high number of inputs or outputs. Nevertheless, this approach ensures that every possible solution is evaluated, which guarantees that the result is optimal regardless the defined optimization goal or cost function.

1) Initialization: As we will see later in Section IV-C3, in each recursive iteration the $\mathbf{H}$ matrix will be extended with a new column, representing the common term that was selected. The extended version of $\mathbf{H}$ will be named $\mathbf{H}^*_{r \times n^*} = \{h^*_{ij}\}$.

In the first iteration we need to work with the original $\mathbf{H}$ matrix, because no common term has been found yet. Therefore we establish $\mathbf{H}^* \leftarrow \mathbf{H}$ and $n^* \leftarrow n$.

It will also be useful to keep track of the number of common terms found so far. We define a new variable called *gates_ct* and set it to 0, as initially there are no common terms.

Given that each tested solution will be compared to the one that is currently considered the best, an initial solution needs to be calculated to serve as a first reference. For this purpose, we will obtain the network that results from applying the MISO algorithm [13] to each output (or row) defined by $\mathbf{H}$. If the optimization parameters under consideration are the ones listed in Section IV-A, they can be calculated with the formulas included in [13]. In any case, these parameters shall be stored in memory for later comparison with other solutions.

2) Search for common terms: The first step of every iteration is to find all the possible groups of two or more terms in one output that are shared at least with another output. This is done by trying all the combinations of 2 or more columns and 2 or more rows, and checking if the elements of $\mathbf{H}^*$ defined by those combinations are equal to 1.

First, we define:

$$\mathcal{P}_{=q}(S) = \big\{P \in \mathcal{P}(S) \big| |P| = q\big\} \tag{4}$$

where $\mathcal{P}(S)$ is the power set (i.e., the set of all subsets) of the set S, and $|P|$ is the cardinality (i.e., the number of elements) of the set P.

We can define Q^R and Q^C as the set of all possible combinations of rows and columns that may together form a common term, respectively:

$$Q^R = \bigcup_{s=2}^{r} \mathcal{P}_{=s}(\mathbb{N}_{\leq r}) \tag{5}$$

$$Q^C = \bigcup_{t=2}^{K} \mathcal{P}_{=t}(\mathbb{N}_{\leq n^*}) \tag{6}$$

We can check if those combinations indeed form a common term by inspecting the $\mathbf{H}^*$ matrix:

$$G = \big\{(A, B) \in Q^R \times Q^C \big| \mathbf{H}^*[A, B] = \mathbf{1}\big\} \tag{7}$$

The result is G: a set of ordered pairs where the elements of the pair are the sets of rows and columns that together form a common term in the current $\mathbf{H}^*$ matrix. In (7), $\mathbf{H}^*[A, B]$ is the submatrix of $\mathbf{H}^*$ that results from selecting the rows indicated by A and the columns indicated by B.

Finding G is a computationally intensive operation, which requires to test a high number of combinations, with their respective memory accesses. For a given $\mathbf{H}^*$, two nested sweeps are needed: one for s ($2 \leq s \leq r$) and another one for t ($2 \leq t \leq n^*$). This implies $\left[\sum_{s=2}^{r} \binom{r}{s}\right] \cdot \left[\sum_{t=2}^{K} \binom{n^*}{t}\right]$ combinations. For each combination, there are $s \cdot t$ accesses to the elements of $\mathbf{H}^*$ in order to check if they are all equal to 1. This means a total of $\left[\sum_{s=2}^{r} \binom{r}{s} s\right] \cdot \left[\sum_{t=2}^{K} \binom{n^*}{t} t\right]$ memory accesses. Note that as n^* increases, the number of tested combinations and memory accesses increase as well. This fact is specially undesirable for our algorithm, because n^* is incremented after each recursive iteration, as will be explained later in Section IV-C3. This issue becomes even more problematic when one takes into account that a new G will have to be obtained for every (A, B) in the current G, reaching several levels of recursion depending on the size and contents of the original $\mathbf{H}$ and also on the value of K. This is just the main problem that will be mitigated later in Section IV-D by means of dynamic programming.

3) Update of the current network: For each found group $(A, B) \in G$, we need to update $\mathbf{H}^*$ so that we can perform a new iteration of the recursive algorithm later. First, the chosen group has to be removed from the current $\mathbf{H}^*$. We do so by filling with zeros the array positions determined by (A, B):

$$h^*_{ij} \leftarrow \begin{cases} 0 & i \in A, j \in B \\ h^*_{ij} & \text{else} \end{cases} \tag{8}$$

Then, a new column is appended to $\mathbf{H}^*$, representing the new common term, which is in fact a FPGA LUT that implements a XOR function with $|A|$ inputs:

$$\mathbf{H}^* \leftarrow [\mathbf{H}^*|\mathbf{c}] \tag{9}$$

with $\mathbf{c} = \{c_i\}$ being a $r \times 1$ column vector such that:

$$c_i = \begin{cases} 0 & i \in A \\ 1 & i \notin A \end{cases} \tag{10}$$

Note that the fact of adding a column to $\mathbf{H}^*$ makes:

$$n^* \leftarrow n^* + 1 \tag{11}$$

Given that such column represents a common term:

$$\text{gates\_ct} \leftarrow \text{gates\_ct} + 1 \tag{12}$$

Also note that the number of elements equal to 1 in $\mathbf{H}^*$ is always reduced with each update, due to the fact that $|B| \geq 2$.

The output of the common term that has just been added needs to be given a level, just like any of the original inputs. It is easy to realize that the level of the new common term is equal to the greatest level of the selected terms (which can be either inputs or common terms) plus 1:

$$l(n^*) \mapsto \max\left\{l(j)|j \in B\right\} + 1 \tag{13}$$

The information about the rows that comprise the new group (A) has already been saved in $\mathbf{H}^*$ with the addition of the new column. But we may also have to save the information about the columns (B) elsewhere for later access. Such information will be needed for reconstructing the resulting network when the algorithm finishes. For this purpose, we can define the function $c(t)$ as the set of selected columns for the tth common term. In each iteration, this function has to be updated with a new element:

$$c(\text{gates\_ct}) \mapsto B \tag{14}$$

4) Parameter calculation: Now that we have updated $\mathbf{H}^*$, n^*, $l(j)$ and gates_ct, the next step is to obtain parameters that characterize the current solution in order to evaluate its quality according to the defined optimization goals. As stated in Section IV-A, the proposed implementation of the algorithm will consider the following optimization parameters: the number of gates, the maximum level, the number of nets and the maximum fan-out.

Such parameters must be calculated for the network that results from applying the MISO algorithm to each of the outputs defined by $\mathbf{H}^*$, so that they correspond to the final network that FS-EDAC would output in the case that it was indeed the best overall one. In spite of being a fast algorithm, applying MISO in each iteration of the FS-EDAC can take a very long time because there is usually a high number of iterations. This section proposes alternative ways for obtaining the optimization parameters without the need to apply the MISO algorithm at all, derived on the equations presented in [13]. This way, the execution time is significantly reduced.

The number of gates of the final network can be easily calculated using the following formula:

$$\text{gates} = \text{gates\_ct} + \sum_{i=1}^{r} \left\lceil \frac{\left[\sum_{j=1}^{n^*} h_{ij}^*\right] - 1}{K-1} \right\rceil \tag{15}$$

The first addend corresponds to the common terms found up to this point. The other one is related to the gates that would be instantiated by the MISO, should it have been applied.

For calculating the maximum level of the final network, we first need to find the number of inputs and common terms that have a level equal to m for a given output i:

$$l_{=m}(i) = \left|\left\{j \in \mathbb{N}_{\leq n^*} \middle| l(j) = m, h_{ij} = 1\right\}\right| \tag{16}$$

The level of the output i of the final network is given by:

$$\text{out\_level}(i) = \left\lceil \log_K \left(\sum_{m=0}^{\infty} l_{=m}(i) \cdot K^m \right) \right\rceil \tag{17}$$

The global maximum level is therefore:

$$\text{max\_level} = \max_i \left\{\text{out\_level}(i)\right\} \tag{18}$$

There is a compact formula for the number of nets of the final network:

$$\text{nets} = \sum_{t=1}^{\text{gates\_ct}} |c(t)| + \sum_{i=1}^{r} \left\lceil K \frac{\left[\sum_{j=1}^{n^*} h_{ij}^*\right] - 1}{K-1} \right\rceil \tag{19}$$

The first addend corresponds to the nets used for connecting the common terms that have been found up to this point, while the other part of the equation refers to the network produced by the MISO algorithm.

For obtaining the maximum fan-out, we first need to obtain the fan-out of every input or common term of the current solution. There is no need to calculate the fan-outs of the gates instantiated by the MISO algorithm, as they are all equal to 1 (that is, the minimum possible value).

The fan-out of the jth column of $\mathbf{H}^*$, which could correspond to either an input or a common term, is simply calculated as:

$$\text{fanout}(j) = \sum_{i=1}^{r} h_{ij}^* \tag{20}$$

Therefore, the maximum fan-out is:

$$\text{max\_fanout} = \max_j \left\{\text{fanout}(j)\right\} \tag{21}$$

5) Solution evaluation: In this step, the solution that is currently considered the best is compared to the one corresponding to this iteration. For this purpose, we will take the optimization parameters calculated in the previous step and apply the procedure defined in Fig. 1, which is used to determine which solution is better.

Following the strategy defined in Section IV-A, the criteria defined in Fig. 1 minimizes the area of the inferred circuit, while trying to maintain good values for the rest of parameters. Nevertheless, any other optimization goal is possible, as long as the associated parameters or cost functions can be obtained or estimated during the synthesis phase.

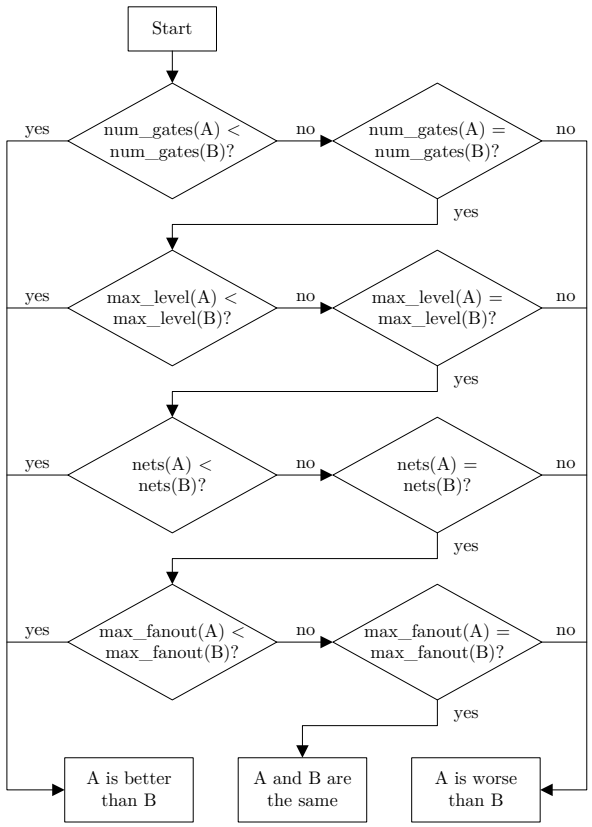

Fig. 1: Priority mechanism to select the best out of two possible solutions.

If this step discovers that the current solution is better than the one that was being considered the best, then the old solution has to be discarded and replaced with the current one. The associated optimization parameters need to be stored in memory for the comparisons that will be performed in later iterations of the algorithm. All the variables that define the new solution ($\mathbf{H}^*$, n^*, $l(j)$, etc) need to be stored as well in case it ends up being the best overall one, so that the final network can be constructed in the last step.

6) Construction of the final network: Once all the groups have been processed, the last step is to apply the MISO algorithm to each output (or row) defined by the $\mathbf{H}^*$ that was considered to be the best among all. The result is a circuit that represents the optimal solution according to the defined optimization criteria.

D. Dynamic programming version

This section describes an alternative implementation of FS-EDAC which is equivalent in terms of functionality to the backtracking version presented in Section IV-C, but considerably faster because it takes advantage of memoization techniques in order to avoid recalculating everything from scratch in each recursive iteration. It does so by either precomputing parameters at the beginning of the algorithm or by reutilizing partial results obtained in previous iterations. In computer science, this type of techniques are usually referred to as *dynamic programming*.

The definition of the steps of this algorithm is the same as in Section IV-C. Therefore, the procedure will be explained in an analogous way, using the same subsection structure.

1) Initialization: First, we get an initial G using (7), just as explained in Section IV-C2. This will be the only time that we will have to apply this computationally intensive operation, as the subsequent Gs will be obtained using a dynamic approach.

In Section IV-C1, we prepared an $\mathbf{H}^*$ matrix that would be updated and extended in every recursive iteration. Now, by means of a custom dynamic programming technique, it will be possible to avoid the need to maintain $\mathbf{H}^*$ at all, thus saving lots of read and write accesses to it. That means that the subsequent searches of groups will have to be performed in an alternative, indirect way. The same idea applies to the calculation of the optimization parameters, which used to depend a lot on $\mathbf{H}^*$ as well.

In spite of not having to maintain $\mathbf{H}^*$, we must keep track of the number of columns that it would have had in the case that it existed. Therefore, we initialize $n^* \leftarrow n$.

In later stages of the algorithm, we will want to know the sum of elements of $\mathbf{H}^*$ equal to 1 at each row. For this purpose, we define a vector $\mathbf{a}_{r\times 1} = \{a_i\}$ that will be updated in each iteration so as to keep track of the row sums:

$$a_i = \sum_{j=1}^{n} h_{ij} \tag{22}$$

Now we prepare the LUTs (software LUTs, not hardware LUTs of the FPGA) that will replace some of the calculations later in the algorithm with simple memory accesses.

There will be a software LUT for the number of gates (or FPGA LUTs) that are needed to merge together a given number of signals s, and an analogue LUT for the number of nets:

$$\text{gates\_LUT}(s) = \left\lceil \frac{s-1}{K-1} \right\rceil, \quad 1 \le s \le n \tag{23}$$

$$\text{nets\_LUT}(s) = \left\lceil K\frac{s-1}{K-1} \right\rceil, \quad 1 \le s \le n \tag{24}$$

There is no need to calculate (23) and (24) for $s > n$, as the worst case would be $s = n$ for any $\mathbf{H}$ with n columns. The worst case happens when all the elements of a row in $\mathbf{H}$ are 1, so all of them are to be merged together in order to produce the output corresponding to that row. Due to the fact that $K \ge 2$, the number of elements in a row equal to 1 will always be reduced with each recursive iteration, so it is impossible to get out of the bounds of the LUTs if they are dimensioned as explained above.

Note that both (23) and (24) could have been obtained by applying the MISO algorithm to an imaginary network with s inputs, but that would have taken more execution time. Also note that those equations do not include the gates and nets that may have been instantiated as a result of having found a number of common terms. We need to prepare separate counters for that purpose: gates_ct and nets_ct, and set their values to 0.

For the levels, we initialize a new matrix $\mathbf{L}_{r\times(m_{max}+1)} = \{l_{im}\}$ with the values given by (16):

$$l_{im} = l_{=m}(i) \tag{25}$$

As usual, the matrix has been dimensioned for the worst case, which occurs when all the elements in two or more rows of $\mathbf{H}$ are equal to 1, all input levels $l(j)$ are equal to each other, and all the common terms are formed by pairs of signals grouped in a stairwise manner. Knowing n and $l(j)$, but not the contents of $\mathbf{H}$ itself, an upper bound for the maximum expected level is:

$$m_{max} = n + \max\left\{l(j) \middle| 1 \leq j \leq n\right\} - 1 \tag{26}$$

Finally, we have to compute the initial fanouts using (20) so as to be able to update them in subsequent steps. The fanouts can be saved in a new $\mathbf{f}_{n^* \times 1} = \{f_j\}$ vector, which will be extended later for every new common term.

2) Search for common terms: At this stage, the set of found groups for the current solution is already located in G, as we calculated it either in the initialization step or in the previous network update step. Therefore, there is no need to perform this time-consuming process any more.

3) Update of the current network: In the backtracking version of the algorithm, this step was devoted to select a group $(A, B) \in G$ and update $\mathbf{H}^*$ accordingly. However, it was explained in Section IV-D1 that we would no longer need to maintain $\mathbf{H}^*$. Instead, now we use a dynamic approach for obtaining the new G that will be used in the next recursive iteration, based on the information currently stored in G.

First of all, we have to apply equations (11), (12) and (14), just as in the non-dynamic version of the algorithm. The row sums will be updated in an incremental way, reflecting the changes that would have been made to $\mathbf{H}^*$ in the case that it existed:

$$a_i \leftarrow \begin{cases} a_i - |B| + 1 & i \in A \\ a_i & \text{else} \end{cases} \tag{27}$$

Now we are ready to start building a brand-new G^*:

$$G^* \leftarrow \emptyset \tag{28}$$

For each $(A', B') \in \{G - (A, B)\}$, we need to calculate a new group to be added to the group set G^*:

$$(A'', B'') = \begin{cases} (A', B') & A' \cap A = \emptyset \vee \\ & B' \cap B = \emptyset \\ (A', (B' - B) \cup n^*) & A' \subseteq A \wedge \\ & B' \cap B \neq \emptyset \wedge \\ & B' \nsubseteq B \\ (\emptyset, \emptyset) & \text{else} \end{cases} \tag{29}$$

We add each new group (A'', B'') to G^* as indicated below:

$$G^* \leftarrow G^* \cup (A'', B'') \tag{30}$$

After all groups $(A', B') \in (G - (A, B))$ have been processed, we can replace the previous G with the one that has just been built:

$$G \leftarrow G^* \tag{31}$$

In the following paragraphs, we will explain equation (29), which reassembles one of the most important contributions that will help to accelerate the algorithm.

$$\mathbf{H}^* = \overset{(A,B)}{\left[\begin{array}{ccccccc} 0 & 0 & 0 & 0 & 0 & \cdots & 1 \\ 0 & 0 & 0 & 1 & 1 & \cdots & 1 \\ 0 & 0 & 0 & 1 & 1 & \cdots & 0 \end{array}\right]}_{(A',B')=(A'',B'')}$$

(a) Independent groups

$$\mathbf{H}^* = \overset{(A,B)}{\left[\begin{array}{ccccc} 0 & 0 & 0 & \cdots & 1 \\ 0 & 0 & 1 & \cdots & 1 \\ 0 & 0 & 1 & \cdots & 1 \end{array}\right]}_{(A',B') \quad (A'',B'')}$$

(b) Updatable group

$$\mathbf{H}^* = \overset{(A,B)}{\left[\begin{array}{ccccc} 0 & 0 & 0 & \cdots & 1 \\ 0 & 0 & 1 & \cdots & 1 \\ 0 & 1 & 1 & \cdots & 0 \end{array}\right]}_{(A',B')}$$

(c) Discardable group

$$\mathbf{H}^* = \overset{(A,B)}{\left[\begin{array}{ccccc} 0 & 0 & 0 & \cdots & 1 \\ 0 & 0 & 1 & \cdots & 1 \\ 0 & 0 & 1 & \cdots & 1 \end{array}\right]}_{(A',B') \quad (A'',B'')}$$

(d) Duplicated group

Fig. 2: Example scenarios when updating G.

The first condition of (29) is the one in which the selected group (A, B) and the group to be updated (A', B') do not have any element in common. This means that the fact of grouping the terms defined by (A, B) does not affect the group (A', B') at all, because they are totally independent. Therefore, we keep (A', B') untouched.

The scenario described in the paragraph above is depicted in Fig. 2a. In this example, $\mathbf{H}^*$ is shown only for illustrative purposes, because in fact it is not calculated. In Fig. 2a, the group (A, B) has been already blanked out as a result of applying (8), and the corresponding common term has been added to the last column of $\mathbf{H}^*$ as in (9). For clarity, the example A, B, A' and B' vectors only contain contiguous elements, but it does not have to be always this way.

The second condition of (29) occurs when there is an overlapping between (A, B) and (A', B') in such a way that a new group can be created using the new common term that was added as a result of having chosen (A, B). Fig. 2b shows an overlapping that has broken the group (A', B'), which can no longer exist in its current form because not all of its elements are equal to 1 ($\mathbf{H}^*[A', B'] \neq 1$). In spite of that, the characteristics of the overlapping allows to define a new group based on (A', B') which uses the recently added common term. In (29), the condition for updating (A', B') has been designed so that $2 \leq |A''| \leq r$ and $2 \leq |B''| \leq K$. Otherwise, the resulting (A'', B'') could be an invalid common term.

In the last case of (29), the group (A', B') is discarded because it has been broken in such a way that it cannot be

updated to use the recently added common term. Fig. 2c shows an example of this last scenario.

It has to be noted that two different (A', B') groups can lead to the same (A'', B'') for a given (A, B) and G. Such is the case of Fig. 2d, if one compares it with Fig. 2b. After applying (30), the fact that G^* is defined as a set avoids any possible duplication.

4) Parameter calculation: In this stage, we will obtain the optimization parameters of the current solution, taking advantage of partial results from previous recursive iterations.

As we have already updated gates_ct in Section IV-D3, we just need to apply the following equation to get the total number of gates:

$$\text{gates} = \text{gates\_ct} + \sum_{i=1}^{r} \text{gates\_LUT}(a_i) \tag{32}$$

The common term selected in Section IV-D3 must be given a level $l(n^*)$ using (13). Then, $\mathbf{L} = \{l_{im}\}$ is updated with:

$$l_{im} \leftarrow \begin{cases} l_{im} + 1 & i \in A, m = l(n^*) \\ l_{im} - 1 & i \in A, m \in \{l(j) | j \in B\} \\ l_{im} & \text{else} \end{cases} \tag{33}$$

Basically, we decrease the level counter of the elements that would have been removed from $\mathbf{H}^*$ in the current recursive iteration, and increase the counter for the level of the new common term. The maximum level is then calculated as in (17) and (18), but replacing $l_{=m}(i)$ with l_{im}.

In order to know the total number of nets, first we need to update nets_ct with the number of nets of the new common term:

$$\text{nets\_ct} \leftarrow \text{nets\_ct} + |B| \tag{34}$$

The number of nets of the final network is calculated as:

$$\text{nets} = \text{nets\_ct} + \sum_{i=1}^{r} \text{nets\_LUT}(a_i) \tag{35}$$

The fan-out vector $\mathbf{f} = \{f_j\}$ is updated according to the following formula:

$$f_j \leftarrow \begin{cases} f_j - |A| + 1 & j \in B \\ f_j & \text{else} \end{cases} \tag{36}$$

After that, the fan-out of the new common term must be appended to $\mathbf{f}$:

$$f_{n^*} \leftarrow |A| \tag{37}$$

The maximum fan-out is obtained with (21), but replacing $\text{fanout}(j)$ with f_j.

E. Greedy variation

This section defines the greedy variant of FS-EDAC, called G-EDAC. The steps of this last algorithm are the same than the ones listed at the beginning of Section IV-C, but the iteration process is different.

In FS-EDAC, Steps 2 to 5 had to be applied to each network obtained in Step 3. With G-EDAC, this new level of recursion is only entered for the network with the best cost function for the current set of common terms under consideration. It is easy to identify the best network because by the time the new recursion level has to be entered, Step 5 would have already been performed for all the networks of the current level, as part of the process of finding the best overall solution.

The process explained above reduces dramatically the number of algorithm branches, and therefore a huge improvement in processing time can be expected. However, the result is not guaranteed to be optimal any more, unlike happened with FS-EDAC. The degree of optimality achieved by FS-EDAC, as well as its processing time, will be evaluated in Section V.

V. Results

In this section, the results of applying the algorithms proposed in this paper will be compared between themselves and to those obtained by commercial synthesis tools, namely *Xilinx Vivado 2014.2*, *Mentor Graphics Precision RTL 2014.09* and *Synopsys Synplify 2013.03*.

Every tested synthesis algorithm or tool has been provided with the same input, although expressed in different forms. The inputs will be passed to FS-EDAC and G-EDAC as explained in Section IV-B. In the case of commercial synthesis tools, the input is usually specified with hardware description languages (HDLs) like VHDL or Verilog. In this work, VHDL has been selected.

A. Execution time

The section is devoted to perform a deep analysis about the execution time of both versions of FS-EDAC: the one based on backtracking and the one based on dynamic programming. The later algorithm was designed for accelerating the first one while maintaining exactly the same functionality and overall structure, which allows to perform a detailed comparison for each algorithm step, finally leading to some interesting conclusions. On the other hand, the same type of analysis cannot be made for G-EDAC as well because the iteration process is not comparable. This is just the reason why the execution time of G-EDAC becomes several order of magnitude faster than any version of FS-EDAC, as we will see later in this section.

The two software implementations of FS-EDAC were implemented in MATLAB, following the equations presented in Sections IV-C and IV-D. The objective of this analysis is to quantify the amount of acceleration obtained by applying dynamic programming techniques to this specific algorithm.

Table I shows the results of the benchmark that was performed in order to compare the two different implementations. The algorithms were executed 10 times in a Intel Core 2 Quad processor at 2.66 GHz, one run after another and forcing them to run in a single core. The time estimates presented in the table are median values of the 10 runs.

The parity-check matrix that was selected as the input for the algorithms was the only possible (15,10) minimum odd-weight EDAC code that can be built according to the method explained in [26]. The size of the associated $\mathbf{H}$ matrix ($n = 15$ and $r = 5$) is big enough for the algorithms to take a reasonable amount of time in being executed, which provides an adequate precision for the benchmark. For this matrix, we will assume that the n inputs come directly from the output

	Backtracking		Dynamic		
	Absolute time (s)	Relative time (%)	Absolute time (s)	Relative time (%)	Acceleration (dir/dyn)
Initialization	0.000	0.000	0.003	0.027	0.048
Search for common terms	33.529	63.278	2.901	23.543	11.558
$\mathbf{H}^*$ building	3.672	6.930	0.000	0.000	∞
Row sums	0.855	1.614	0.416	3.376	2.055
Number of gates	0.224	0.423	0.207	1.680	1.082
Maximum level	3.390	6.398	2.522	20.467	1.344
Number of nets	0.577	1.089	0.206	1.672	2.801
Maximum fan-out	0.689	1.300	0.344	2.792	2.003
Globals declaration	6.577	12.412	3.601	29.223	1.826
Function overhead	3.237	6.109	1.885	15.297	1.717
Solution evaluation	0.237	0.447	0.237	1.923	1.000
Total	52.987	100.000	12.322	100.000	4.300

TABLE I: Execution time breakdown for the two software implementations with an example $\mathbf{H}$

of n flip-flops, so $l(1 \leq j \leq n) = 0$. The maximum number of inputs per LUT will be set to $K = 3$.

In Table I, the first column lists the different tasks in which the software implementations spend a measurable CPU time, including the total execution time. Most of the listed tasks are directly related to the different steps of the algorithm as described in Sections IV-C and IV-D. Others, like *Globals declaration* and *Function overhead*, are software-specific issues, not closely related with the algorithm itself.

The first and most important conclusion that can be extracted from Table I is that, indeed, the dynamic programming techniques have permitted to accelerate the total execution of the algorithm. With the example inputs that were established, the measured acceleration factor was 4.3.

A closer inspection of Table I reveals that every single step of the algorithm is accelerated, with the exception of the *Initialization* step, which is something that could be expected because a full search of groups in the initial $\mathbf{H}$ has to be performed in the dynamic version of the algorithm. However, the impact of this delay in the total execution time is insignificant, as this is a task that is executed only once. The rest of tasks listed in Table I are executed once for every possible solution. With $\mathbf{H}$ under consideration, the number of tested solutions is 236321 in both versions of the algorithm, as they are functionally equivalent.

Table I shows that the step which has benefited most from the dynamic approach is the search for common terms. Also, it is interesting to note that the dynamic version does not spend any time in building the $\mathbf{H}^*$ matrix. This behaviour is consistent to what was explained in Section IV-D1 about the non-maintenance of $\mathbf{H}^*$.

Up to this point, only the *Absolute time* and *Acceleration* columns have been analysed. But there are also many interesting conclusions that can be extracted from the *Relative time* columns. With them, we can discover which tasks have a greater impact on the total execution time, and think of some ideas to make them faster.

The relative time spent in the *Globals declaration* entry shows that it is the main component that is decelerating the dynamic algorithm. This component can be eliminated by using another programming language that makes an efficient handling of the global variables, with no unwanted delays because of context changes. Such is the case of the C/C++ language, for example. If we removed the row *Globals declaration* from Table I, we would get a overall acceleration of $\times 5.321$. A proper C/C++ implementation would also decrease the execution time of every single algorithm stage.

The second most important component in the execution time of the dynamic implementation is the *Search for common terms* step. After all the optimizations that have been made, the only effective way of reducing the time spent at this stage would be to reduce the number of tested groups, in such a way that the best solution is not discarded.

Further analysis of the 236321 tested solutions reveals that there are only 35 unique solutions. The rest of them can be considered duplicates, with exactly the same structure as other solutions but combining different inputs, which in the end makes no difference at all. If the algorithm had been able to discard those 236286 duplicates, its execution time would be dramatically reduced, getting as fast as a few milliseconds for the present benchmark considering the fact that the execution time is proportional to the number of tested cases. Note that such huge boost would be possible because of the regularity of the selected $\mathbf{H}$ matrix, which contributes to increase the number of duplicates. Other $\mathbf{H}$ matrices can be less regular, but still, processing speed will surely be reduced if the duplicates are identified in an efficient way.

Another way to accelerate the algorithm is to change the strategy and follow a greedy approach like G-EDAC. However, this type of techniques do not always ensure the best possible solution, and sometimes they have to be completely redesigned if the optimization parameters or their priority are changed. By avoiding duplicate solutions in brute-force implementations, as explained in the paragraph above, one would theoretically be able to get speeds comparable to those of a greedy algorithm and still obtain optimal results, keeping the flexibility to alter the optimization parameters at will. As a reference, G-EDAC was made to process the same $\mathbf{H}$ of the present benchmark, and the resulting processing time was 190 milliseconds.

B. Hardware implementation results

In this section, the hardware implementation results of the algorithms described in this paper will be analysed and

TABLE II: Post place and route results of the Hsiao (22,16) function in an A3P030 FPGA ($K = 3$)

Synthesis tool	LUTs	Max. level	Nets	Max. fan-out	Common terms	Max path delay (ns)	Total power (mW)
Synplify	29	3	76	3	1	3.640	1.961
Prec-RTL	28	3	74	3	2	3.551	1.737
MISO	24	2	72	3	0	3.099	1.660
G-EDAC	20	2	60	3	4	3.052	1.433
FS-EDAC	19	2	57	3	5	2.995	1.287

compare to those obtained by several commercial synthesis tools. Additionally, the standalone version of MISO will be used for processing a full EDAC function, not just as a part of the present algorithms. For an explanation about how to apply MISO to a multi-output function like EDAC, please see [13].

The EDAC functions will be passed to the commercial synthesis tools as VHDL source files containing the defining equations with no further manipulations. For example, the (8,4) SEC-DED code defined in equation (3) would be translated to VHDL as follows:

```
s(1) <= m(1) xor m(2) xor m(3) xor m(5);
s(2) <= m(1) xor m(2) xor m(4) xor m(6);
s(3) <= m(1) xor m(3) xor m(4) xor m(7);
s(4) <= m(2) xor m(3) xor m(4) xor m(8);
```

Signals `m` and `s` are connected to a set of flip-flops, which in turn are connected to the input and output ports of the top level entity, so they are associated with physical pins of the FPGA. According to the nomenclature presented in this paper, this means that all the inputs will have a level equal to 0. Given that no registers have been instantiated between signals `m` and `s`, the core of the circuit will be purely combinational.

When a synthesis tool is given a piece of VHDL code like the one shown above, it generates a netlist in EDIF format which is specific for the target technology. The netlist defines a circuit that can only contain the components available in the target FPGA, without specifying where they will be placed or how the connections will be routed throughout the die. Those tasks are in charge of the place and route tools provided by the FPGA vendor (available as part of *Libero SoC* for *Microsemi* FPGAs or as part of *Vivado* for *Xilinx* FPGAs).

FS-EDAC produces a simplified circuit, but such circuit has to be inserted somehow in the design flow, so that the place and route tools can finish implementing it. One possible way is to generate an EDIF netlist, emulating a normal synthesis tool. However, the authors chose to generate a technology-dependent VHDL file, which is a VHDL file that includes specific libraries for the target technology and only instantiates components that are present in such technology. Using this technique, the generated VHDL cores can be easily integrated in larger designs by adding them into the design flow just like any other HDL source file. If the VHDL file is defined correctly, a commercial synthesis tool like Synplify or Vivado will not modify the circuits defined in those files. Instead, it will perform a direct translation from VHDL to EDIF.

Several tests have been executed for this study, evaluating the selected synthesis tools for two different FPGA architectures: one from Microsemi and the other from Xilinx.

The synthesis, place and route tools have been configured to reduce the area utilization as much as possible, so as to match the first optimization objective that was established for FS-EDAC (see Section IV-A and Fig. 1). For this purpose, the only tool that needed a change of the default options was Precision RTL, whose *Optimization goal* property was set to *Compile for area*. On the other hand, the techniques that were applied in order to obtain the best possible timing performances (timing was a lower-priority optimization goal) will be explained in subsequent subsections, because each tool has its own way to do so. The results shown in the tables of this section have been extracted from the reports generated by the respective tools, after completing the process of placing and routing the synthesized circuits.

The maximum path delay is measured from the input to the output of the EDAC entity. Each input and output signal of this combinational block has been connected to one flip-flop so as to minimize the effects of widespread placing, which happens when the EDAC entity is directly connected to the FPGA I/O pins. The existence of the flip-flops allows the placing tools to make the circuits more compact, so the comparison of the maximum path delay is more fair. Besides, the results will be more similar to those that would be found in a real application, as it is a usual practice to register the inputs and outputs of every FPGA core.

In the present study, the total power (static power plus dynamic power) has been calculated as accurately as possible, feeding the power estimation tools of the respective manufactures with the output of a post place and route simulation of the circuit. The test bench switches the inputs at every clock cycle with a probability of 50%. On the other hand, the clock frequency has been established to 100 MHz. The computed total power only includes the elements that belong to the EDAC entity itself (i.e., the consumption of clock nets, I/O blocks and flip-flops are not considered).

1) Results with a 3-input LUT FPGA: In the first set of tests, the selected FPGA model is an A3P030, from the Microsemi ProASIC3 family. The core of this FPGA consists of a sea of cells called *VersaTiles*. Each cell can be configured either as a 3-input LUT ($K = 3$), a D-flip-flop or a latch, and they may be connected between themselves through any of the four levels of routing hierarchy.

The results obtained for the A3P030 are specially relevant, as the ProASIC3 family is widely used for developing and prototyping FPGA designs for space missions [2], [19], [20], [33], [34], [44], where EDAC functions are usually implemented for protecting memories from SEU. Once the designs have been finished, the ProASIC3 netlists can be exported to a RTAX FPGA. This latter family of FPGAs is very popular in space

TABLE III: Post place and route results of the Hsiao (72,64) function in an XC7A100T FPGA ($K = 6$)

Synthesis tool	LUTs	Max. level	Nets	Max. fan-out	Common terms	Max path delay (ns)	Total power (mW)	Sensitive bits
Vivado	48	3	256	5	0	2.580	1.404	23222
Synplify	52	2	260	5	0	2.486	1.496	23824
Prec-RTL	47	2	254	5	1	2.519	1.445	23388
MISO	48	2	256	5	0	2.476	1.464	23284
G-EDAC	35	2	192	3	12	2.332	1.103	21223

applications due to their high reliability against radiation.

Because of it being flash-based, it is not possible to re-program a ProASIC3 in run-time, so the EDAC cores cannot be used to protect the configuration memory of the FPGA itself. Still, the use of the proposed algorithms will be beneficial for the EDAC cores and for the reliability of the systems as a whole because of the reasons outlined in Section I.

All the algorithms under study where fed with the (22,16) EDAC function presented in [26]. Table II shows that the proposed algorithms (G-EDAC and FS-EDAC, in any of its versions) provide better results than Synplify and Precision RTL for every evaluated parameter with the exception of the maximum fan-out, which the same for all the tested tools. The performance of Vivado could not be evaluated in this case because this tool does not support FPGAs by Microsemi.

Since the FS-EDAC algorithm tries all possible combinations of common terms among every output, it is natural that it produces a circuit with less area than any other algorithm, as it was configured to find the solution with the fewer number of LUTs. In this case, FS-EDAC obtains around $^1/_3$ less LUTs than the commercial synthesis tools.

It is specially relevant the fact that G-EDAC produces very similar results to FS-EDAC in spite of being several orders of magnitude faster. Still, it has to be checked in the next subsection whether such degree of optimality is also achievable with bigger input designs and different FPGA architectures.

It can happen that FS-EDAC and G-EDAC get a high value in the maximum level, as this parameter has been given a lower optimization priority than the number of LUTs. However, Table II shows that both FS-EDAC and G-EDAC have outperformed the rest of algorithms for this parameter as well. One could think that the higher number of common terms has allowed this behaviour, however, the truth is that it all depends on how well these terms are grouped. All of the proposed algorithms take a great care in grouping the common terms properly.

It is surprising to realize that the MISO algorithm obtains better results than other commercial tools for this case, in spite of being a simple algorithm which processes each output separately, completing its execution in a matter of milliseconds even for the biggest designs. According to the *Common terms* column, both Synplify and Precision RTL have processed the EDAC function as a whole (therefore identifying common terms), but still, the area utilization is worse than MISO.

The seventh column of Table II confirms that the assumption about the relationship between the maximum level and the maximum path delay has been correct. Assuming that the place and route tools perform well, the circuits with lower maximum level are prone to be faster, as explained in Section IV-A. In order to extract the maximum effort of the Libero IDE place and route tools, a multi-pass place and route process was performed with a strict clock period constraint (1 nanosecond). After 20 passes, the solution with lower maximum path delay was selected. In this case, the speed improvement of the proposed algorithms has been between 14.05% and 17.72% when compared to the commercial synthesis tools.

The estimated total power has proven to be approximately proportional to the number of used resources, that is, the number of LUTs and nets. The improvement is equal to 34.37% for the most favorable case. Nevertheless, it is important to note that the resulting total power also depends a lot on the performance of the place and route tools, just like happens with the maximum path delay. It could occur, for example, that an inferred net goes through a lot of FPGA nets and switching matrices or multiplexers, resulting in longer path delays and increased total power consumption. These effects are minimized by using the technique explained in the previous paragraph, which has allowed to obtain consistent results.

2) Results with a 6-input LUT FPGA: The other battery of tests was obtained with the Vivado software suite, using the XC7A100T, a 6-input LUT FPGA ($K = 6$) from the Artix-7 family. The architecture of this family is very different from the ProASIC3, and the differences are not limited to the number of inputs of each LUT. The main logic resources of this FPGA are the *Configurable Logic Blocks* (CLBs). Each one consists on a pair of slices, and every slice contains four 6-input LUTs plus four storage elements (latches or flip-flops). The CLBs are arranged in a regular array of rows and columns, connected between them by means of switching matrices and general-routing resources that run vertically and horizontally around the CLBs. The configuration memory of this FPGA is stored in a SRAM memory, which is specially sensitive to SEU, so it will be highly interesting to check the increase of reliability of the EDAC cores themselves when the proposed synthesis algorithms are applied.

In this section, the algorithms where fed with a larger EDAC function than in Section V-B1, namely the (72,64) EDAC function presented in [26]. It was considered convenient to choose a bigger input function so as to allow some margin of optimization to the synthesis tools now that K is also higher than before. A general rule is that a higher value of K usually allows to infer smaller circuits, given that each LUT will be able to host a bigger portion of the complete logic function. Having smaller synthesized circuits would not permit to draw firm conclusions, as the results would be more similar between themselves.

FS-EDAC could not be tested in this new analysis because the execution times where unacceptably long. An estimation showed that it could take several years to execute FS-EDAC for the EDAC function under consideration, in contrast to the 9.33 seconds taken by G-EDAC to process the same function.

Table III shows the results for the XC7A100T. G-EDAC is clearly the algorithm that performs best, in spite of the fact that greedy approaches do not always guarantee optimal solutions. The reduction of area utilization with the other tools is now between 25.53% and 32.69%. Regarding the maximum level, only Vivado has obtained a value of 3, which explains the worse value in the maximum path delay for this tool (9.61% higher than G-EDAC). For the rest of the tools, the maximum path delays are more similar between themselves, as all of them have obtained a maximum level equal to 2.

Neither Synplify nor Vivado have been able to find any common term, and still, MISO obtains better figures of merit than the other two without even trying to find a common term. This is because MISO guarantees that the solution minimizes all the optimization parameters defined in Section IV-A at the same time, as long as LUT output sharing is not allowed (i.e., no common terms). In this case, Precision RTL has beaten MISO because the first one has grouped a proper common term, although the mere fact of selecting one or more common terms does not always mean an improvement over selecting none (see Table II).

It can be seen in Table III that the relationship between the post-place and route maximum path delay and the total power with the optimization parameters is not as strong as in Table II. This is because the method of running multiple passes is not possible in Vivado, as its place and route algorithms seem to be deterministic (i.e., they do not rely on random seeds). In order to extract the maximum effort from this tool, the chosen strategy was to iteratively reduce the clock period constraint until such constraint cannot be met by Vivado.

Lastly, an additional column was appended to Table III, which was not present in Table II. This new column gives the number of sensitive bits calculated by the Vivado bitstream generation tool. Unfortunately, such information is not reported by Libero SoC, so it could not be included in Table II.

Table III confirms the idea that a lower area usage results in a smaller number of sensitive bits. The reduction obtained with G-EDAC with respect to the other algorithms and tools is between 8.61% and 10.91%. This is just the percentage of improvement in circuit reliability that can be expected for an EDAC core implemented in a SRAM-based FPGA, as the number of sensitive bits is directly proportional to the MTBF (Mean Time Between Failures) for a given radiation environment and FPGA model. This means that the ultimate goal of improving the reliability of EDAC cores implemented in SRAM FPGAs has been successfully accomplished.

VI. Conclusions

In this paper, three algorithms specialized in synthesizing EDAC functions on FPGAs have been extensively described and evaluated. By exploiting the properties of this particular problem, the algorithms are able to obtain optimal (FS-EDAC) or quasi-optimal (G-EDAC) solutions given a predefined optimization criteria.

When compared to the results obtained by means of commercial synthesis tools, the proposed algorithms can reduce the area, maximum path delay and power consumption of the resulting circuits up to 34.48%, 17.72% and 34.37%, respectively. These numbers confirm the fact that there is still a wide margin of improvement for the synthesis tools that are currently available in commercial FPGA design suites.

In addition, the EDAC cores were effectivily hardened against radiation, achieving an increase of 10.91% in the MTBF. The higher reliability becomes even more useful in the cases where the EDAC cores are in charge of protecting the configuration memory that defines the circuitry implemented in the FPGA, including the EDAC cores themselves.

Even though the algorithms proposed in this paper were designed specifically for FPGA devices, it is easy to adapt FS-EDAC so that it minimizes other parameters that are specific to ASIC devices, e.g., the number of transistors and/or the die area. On the other hand, the greedy version of the algorithm (G-EDAC) is also adaptable, but a new battery of tests should be executed in order to evaluate its degree of optimality.

Being a highly recursive algorithm, FS-EDAC has the drawback that it is not practical for dealing with large functions, as the number of possible combinations increases exponentially with every added input or output. It was shown in Section V-A that dynamic programming techniques produce a significant performance boost (5 times faster than backtracking), but still, the long processing times prevented the authors from determining the optimal results for the largest tested circuit. As future work, it will be analysed whether more intelligent searches can be performed for finding common terms, as the current implementations try many combinations that are identical between them (see Section V-A). A smarter algorithm which is able to skip some or all of the duplicates may have a reasonable execution time even for the largest functions.

For all the designed algorithms, the MATLAB language and software suite was selected because it allowed fast development and easy debugging, reducing the risk of programming errors. However, it was said in Section V-A that C/C++ implementations would be very beneficial for accelerating the algorithms. Software ports to the C/C++ language will consequently be scheduled as future work for this paper.

Finally, another line of research could be opened regarding the implementation of heuristic or greedy algorithms more sophisticated than G-EDAC. Again, having developed FS-EDAC will be useful for this task, as the degree of optimality will be subject to be measured by comparing the results of the new algorithms with the ones obtained by FS-EDAC.

Acknowledgements

This work was supported by the Spanish Ministry of Educación, Cultura y Deporte under the grant FPU12/05573, and by the Spanish Ministry of Economía project ESP2013-48362-C2-2-P, in the frame of the activities of the Instrument Control Unit of the Infrarred Instrument of the ESA Euclid Mission carried out by the Dept. of Electronics and Computer Technology of the Universidad Politécnica de Cartagena.

REFERENCES

[1] ADELL, P., WITULSKI, A., SCHRIMPF, R., BARONTI, F., HOLMAN, W., AND GALLOWAY, K. Digital control for radiation-hardened switching converters in space. *Aerospace and Electronic Systems, IEEE Transactions on 46*, 2 (April 2010), 761–770.

[2] ALFONZO, M., FRAIRE, J., KOCIAN, E., AND ALVAREZ, N. Development of a dtn bundle protocol convergence layer for spacewire. In *Biennial Congress of Argentina (ARGENCON), 2014 IEEE* (June 2014), pp. 770–775.

[3] ALMEIDA, G., BEZERRA, E., CARGNINI, L., FAGUNDES, R., AND MESQUITA, D. A reed-solomon algorithm for fpga area optimization in space applications. In *Adaptive Hardware and Systems, 2007. AHS 2007. Second NASA/ESA Conference on* (Aug 2007), pp. 243–249.

[4] AYMEN, F., BELGACEM, H., AND CHIRAZ, K. A new efficient self-checking hsiao sec-ded memory error correcting code. In *Microelectronics (ICM), 2011 International Conference on* (2011), pp. 1–5.

[5] BARTH, J. L., DYER, C. S., AND STASSINOPOULOS, E. G. Space, atmospheric and terrestrial radiation environments. *Nuclear Science, IEEE Transactions on 50*, 3 (jun. 2003), 466–482.

[6] BENTOUTOU, Y. A real time edac system for applications onboard earth observation small satellites. *Aerospace and Electronic Systems, IEEE Transactions on 48*, 1 (2012), 648–657.

[7] CARDARILLI, G., OTTAVI, M., PONTARELLI, S., RE, M., AND SALSANO, A. Fault tolerant solid state mass memory for space applications. *Aerospace and Electronic Systems, IEEE Transactions on 41*, 4 (Oct 2005), 1353–1372.

[8] CHANDRA, V., AND AITKEN, R. Impact of technology and voltage scaling on the soft error susceptibility in nanoscale CMOS. In *Defect and Fault Tolerance of VLSI Systems, 2008. DFTVS '08. IEEE International Symposium on* (2008), pp. 114–122.

[9] CHEN, D., AND CONG, J. Daomap: a depth-optimal area optimization mapping algorithm for fpga designs. In *Computer Aided Design, 2004. ICCAD-2004. IEEE/ACM International Conference on* (Nov 2004), pp. 752–759.

[10] CHEN, G., AND CONG, J. Simultaneous logic decomposition with technology mapping in fpga designs. In *Proceedings of the 2001 ACM/SIGDA Ninth International Symposium on Field Programmable Gate Arrays* (New York, NY, USA, 2001), FPGA '01, ACM, pp. 48–55.

[11] CHEN, K.-C., CONG, J., DING, Y., KAHNG, A., AND TRAJMAR, P. Dag-map: graph-based fpga technology mapping for delay optimization. *Design Test of Computers, IEEE 9*, 3 (Sept 1992), 7–20.

[12] CHEN, P.-Y., YEH, Y.-T., CHEN, C.-H., YEH, J.-C., WU, C.-W., LEE, J.-S., AND LIN, Y.-C. An enhanced edac methodology for low power psram. In *Test Conference, 2006. ITC '06. IEEE International* (2006), pp. 1–10.

[13] COLODRO-CONDE, C., AND TOLEDO-MOREO, R. Improving the implementation of edac functions in radiation-hardened fpgas. In *FPGAs and Parallel Architectures for Aerospace Applications*. Springer, 2014, p. TO BE PUBLISHED.

[14] CONG, J., AND HWANG, Y.-Y. Structural gate decomposition for depth-optimal technology mapping in lut-based fpga design. In *Design Automation Conference Proceedings 1996, 33rd* (Jun 1996), pp. 726–729.

[15] CONG, J., AND MINKOVICH, K. Optimality study of logic synthesis for lut-based fpgas. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on 26*, 2 (Feb 2007), 230–239.

[16] CONG, J., WU, C., AND DING, Y. Cut ranking and pruning: Enabling a general and efficient fpga mapping solution. In *Proceedings of the 1999 ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays* (New York, NY, USA, 1999), FPGA '99, ACM, pp. 29–35.

[17] CORY, W. E., SCHULTZ, D. P., AND YOUNG, S. P. Error checking parity and syndrome of a block of data with relocated parity bits, Sept. 16 2008. US Patent 7,426,678.

[18] DODD, P., AND MASSENGILL, L. Basic mechanisms and modeling of single-event upset in digital microelectronics. *Nuclear Science, IEEE Transactions on 50*, 3 (June 2003), 583–602.

[19] DUDNIK, O., PRIETO, M., KURBATOV, E., SANCHEZ, S., TITOV, K., SYLWESTER, J., GBUREK, S., AND PODGÓRSKI, P. Functional capabilities of the breadboard model of sidra satellite-borne instrument. *Problems of Atomic Science and Technology*, 3 (2013), 289–296.

[20] FRAIRE, J., MADOERY, P., AND FINOCHIETTO, J. Leveraging routing performance and congestion avoidance in predictable delay tolerant networks. In *Wireless for Space and Extreme Environments (WiSEE), 2014 IEEE International Conference on* (Oct 2014), pp. 1–7.

[21] FUJIWARA, E. *Code Design for Dependable Systems: Theory and Practical Applications*. Wiley, 2006.

[22] FULLER, E., CAFFREY, M., SALAZAR, A., CARMICHAEL, C., AND FABULA, J. Radiation testing update, seu mitigation, and availability analysis of the virtex fpga for space re-configurable computing. In *in Proc. International Conference on Military and Aerospace Programmable Logic Devices* (2000).

[23] GAO, W., AND SIMMONS, S. A study on the vlsi implementation of ecc for embedded dram. In *Electrical and Computer Engineering, 2003. IEEE CCECE 2003. Canadian Conference on* (2003), vol. 1, pp. 203–206 vol.1.

[24] HAO, L., AND YU, L. A study on the hardware implementation of edac. In *Convergence and Hybrid Information Technology, 2008. ICCIT '08. Third International Conference on* (2008), vol. 2, pp. 222–225.

[25] HEINER, J., COLLINS, N., AND WIRTHLIN, M. Fault tolerant icap controller for high-reliable internal scrubbing. In *Aerospace Conference, 2008 IEEE* (March 2008), pp. 1–10.

[26] HSIAO, M. A class of optimal minimum odd-weight-column sec-ded codes. *IBM Journal of Research and Development 14*, 4 (1970), 395–401.

[27] JOHANSSON, R. Two error-detecting and correcting circuits for space applications. In *Proceedings of the The Twenty-Sixth Annual International Symposium on Fault-Tolerant Computing (FTCS '96)* (Washington, DC, USA, 1996), FTCS '96, IEEE Computer Society, pp. 436–439.

[28] JULIATO, M., AND GEBOTYS, C. A quantitative analysis of a novel seu-resistant sha-2 and hmac architecture for space missions security. *Aerospace and Electronic Systems, IEEE Transactions on 49*, 3 (July 2013), 1536–1554.

[29] LANUZZA, M., ZICARI, P., FRUSTACI, F., PERRI, S., AND CORSONELLO, P. An efficient and low-cost design methodology to improve sram-based fpga robustness in space and avionics applications. In *Proceedings of the 5th International Workshop on Reconfigurable Computing: Architectures, Tools and Applications* (Berlin, Heidelberg, 2009), ARC '09, Springer-Verlag, pp. 74–84.

[30] LANUZZA, M., ZICARI, P., FRUSTACI, F., PERRI, S., AND CORSONELLO, P. A self-hosting configuration management system to mitigate the impact of radiation-induced multi-bit upsets in sram-based fpgas. In *Industrial Electronics (ISIE), 2010 IEEE International Symposium on* (July 2010), pp. 1989–1994.

[31] LING, A., SINGH, D., AND BROWN, S. Fpga technology mapping: a study of optimality. In *Design Automation Conference, 2005. Proceedings. 42nd* (June 2005), pp. 427–432.

[32] MAHATME, N., JAGANNATHAN, S., LOVELESS, T., MASSENGILL, L., BHUVA, B., WEN, S. J., AND WONG, R. Comparison of combinational and sequential error rates for a deep submicron process. *Nuclear Science, IEEE Transactions on 58*, 6 (2011), 2719–2725.

[33] MELNIKOVA, O. Overview of the prototyping technologies for actel®rtax-s fpgas. In *Design Test Symposium (EWDTS), 2011 9th East-West* (Sept 2011), pp. 90–93.

[34] MILLARD, W., AND HASKINS, C. Digital signal processing architecture design for gate array based software defined radios. In *Aerospace Conference, 2012 IEEE* (March 2012), pp. 1–20.

[35] MISHCHENKO, A., CHATTERJEE, S., AND BRAYTON, R. Improvements to technology mapping for lut-based fpgas. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on 26*, 2 (Feb 2007), 240–253.

[36] MORGAN, K. S., CAFFREY, M., CARROLL, J., GIBELYOU, D., GRAHAM, P., HOWES, W., JOHNSON, J., MCMURTREY, D., OSTLER, P., PRATT, B., QUINN, H., AND WIRTHLIN, M. Fault tolerance techniques and reliability modeling for sram-based fpgas. *Radiation Effects in Semiconductors* (2010), 249.

[37] NEUBERGER, G., DE LIMA KASTENSMIDT, F., AND REIS, R. An automatic technique for optimizing reed-solomon codes to improve fault tolerance in memories. *Design Test of Computers, IEEE 22*, 1 (Jan 2005), 50–58.

[38] OSTLER, P., CAFFREY, M., GIBELYOU, D., GRAHAM, P., MORGAN, K., PRATT, B., QUINN, H., AND WIRTHLIN, M. Sram fpga reliability analysis for harsh radiation environments. *Nuclear Science, IEEE Transactions on 56*, 6 (Dec 2009), 3519–3526.

[39] PATIL, N., DAS, D., SCANFF, E., AND PECHT, M. Long term storage reliability of antifuse field programmable gate arrays. *Microelectronics Reliability 53*, 12 (2013), 2052 – 2056.

[40] REED, I. S., AND SOLOMON, G. Polynomial codes over certain finite fields. *Journal of the Society for Industrial & Applied Mathematics 8*, 2 (1960), 300–304.

[41] SUNDARARAJAN, P., CARMICHAEL, C., MCMILLAN, S., BLODGET, B., AND PATTERSON, C. Methods of estimating susceptibility to single

event upsets for a design implemented in an fpga, July 24 2007. US Patent 7,249,010.

[42] VIOLANTE, M., STERPONE, L., CESCHIA, M., BORTOLATO, D., BERNARDI, P., REORDA, M., AND PACCAGNELLA, A. Simulation-based analysis of seu effects in sram-based fpgas. *Nuclear Science, IEEE Transactions on 51*, 6 (Dec 2004), 3354–3359.

[43] WIRTHLIN, M. J., TAKAI, H., AND HARDING, A. Soft error rate estimations of the kintex-7 fpga within the atlas liquid argon (lar) calorimeter. *Journal of Instrumentation 9*, 01 (2014), C01025.

[44] ZANE, S., WALTON, D., KENNEDY, T., FEROCI, M., DEN HERDER, J.-W., AHANGARIANABHARI, M., ARGAN, A., AZZARELLO, P., BALDAZZI, G., BARBERA, M., ET AL. The large area detector of loft: the large observatory for x-ray timing. In *Proc. SPIE* (2014), vol. 9144, pp. 91442W–91442W–19.

Carlos Colodro-Conde received the Telecommunications Engineering degree from the Universidad Politécnica de Cartagena (UPCT) in 2011. He is currently a Ph.D. student at the Department of Electronics and Computer Technology, Universidad Politécnica de Cartagena. His research is focused on instrument control electronics for space missions and ground telescopes.

Rafael Toledo-Moreo received the M.S. degree in automation and electronics engineering from the Technical University of Cartagena (UPCT), Cartagena, Spain, in 2002 and the Ph.D. degree in computer science from the University of Murcia (UMU), Murcia, Spain, in 2006. Between 2008 and 2010 he worked for 1 year at the Laboratoire Central des Ponts et Chausses (currently IFSTTAR), Nantes, France, and he was a guest researcher at the University of Paris South XI, France. He is currently an Associate Professor with the Department of Electronics and Computer Technology at UPCT, Academic Secretary of the School of Telecommunication of Engineering, and the Head of the Telefnica Chair. Dr. Toledo-Moreo is the PI of the infrared instrument control unit of the Euclid mission, an m-class cosmological mission of ESA. His main fields of interest are space electronics and intelligent transportation systems. In these domains, he has participated in more than 30 research projects and published more than 70 research articles. He is also an Associate Editor and member of International Program Committees of several journals and conferences related to intelligent transportation systems.

LIST OF FIGURES