A Proposal of an ECC-based Adaptive Fault-Tolerant Mechanism for 16-bit data words

J. Gracia-Morán (ORCID), L. J. Saiz-Adalid (ORCID), J. C. Baraza-Calvo (ORCID), D. Gil-Tomás (ORCID), P. J. Gil-Vicente (ORCID), *Member, IEEE*

***Abstract*— Actual memory systems provide large storage capacity thanks to the integration scale level achieved in CMOS technology. This increment in storage capacity comes with an augment on their fault rate. In this way, the probability of experiencing Single or Multiple Cell Upsets has risen. Error Correction Codes (ECC) are a fault-tolerant mechanism broadly employed to protect memory systems. Usually, an ECC-based fault tolerance mechanism is designed with fixed correction and detection capabilities. However, in some contexts, current memory systems can suffer a variable fault rate during their operation. Thus, it seems very interesting that this fault-tolerant mechanism would be able to adapt to these variable fault conditions.**

This work proposes an Adaptive Fault-Tolerant mechanism based on ECC. This mechanism can adapt to different fault conditions, being able to correct and/or detect single and multiple bits in error. The Adaptive Fault-Tolerant mechanism proposed uses a unique encoder and various decoders. Therefore, there is no need to re-encode the data to change the error coverage since the unique encoder and the equal redundancy are the same regardless of the fault tolerance required. In addition, we have studied the area, delay, and power consumption overheads produced by the inclusion of the redundant bits, the encoder, and the decoders of the ECC in a computer system.

***Index Terms*—Adaptability, Error Correction Codes, Fault Tolerance, Multiple Bit Errors, Single Bit Errors, Reliability**

I. Introducción

Durante estos últimos años, el desarrollo de la tecnología CMOS ha permitido la implementación de sistemas de memoria que han incrementado su capacidad de almacenamiento hasta niveles impensables hace pocos años, pero a costa de aumentar su tasa de fallos [1]. En este sentido, la carga crítica de una celda de memoria, así como la energía necesaria para causar la alteración de una única celda (efecto conocido como *Single Cell Upset*) también se han reducido, de tal forma que el impacto de una única partícula también puede provocar la alteración de varias celdas de memoria (efecto conocido como *Multiple Cell Upsets*) [2][3][4].

Los códigos de corrección de errores (ECC, del inglés *Error Correction Codes*) son uno de los mecanismos más utilizados para proteger los sistemas de memoria. Este mecanismo de tolerancia a fallos es capaz de detectar y/o corregir bits erróneos. La detección implica que el ECC sabe que hay bits erróneos, pero no puede determinar cuáles son. Si es posible localizar los errores, se pueden corregir invirtiendo los bits afectados. Tradicionalmente, se han usado como protección los códigos SEC (*Single Error Correction*, capaces de corregir un error de un bit) o los códigos SEC-DED (*Single Error Correction – Double Error Detection*, que pueden corregir un error de un bit, o detectar hasta dos errores en dos bits independientes dentro del mismo dato) [5][6][7]. En aplicaciones críticas, en las que se espera la ocurrencia de fallos múltiples, se pueden utilizar otros ECC más complejos y con una mayor tolerancia a fallos [8][9][10].

Al introducir un ECC en un sistema de memoria, se emplean circuitos de codificación y decodificación que provocan una serie de sobrecargas no deseadas, tales como el incremento en el área de silicio ocupada, el aumento del retardo en la lectura/escritura de los datos, y un mayor consumo de energía. Además, hay que añadir a cada palabra de datos varios bits adicionales (llamados de código, de paridad o redundantes), necesarios para detectar y/o corregir los posibles errores ocurridos, y que también se tienen que almacenar.

Normalmente, los ECC se diseñan con una capacidad de tolerancia a fallos fija, basando esta capacidad en el peor caso posible. Este diseño provoca una gran cantidad de operaciones que no son necesarias, lo que implica un mayor consumo de energía y un aumento en el retardo del circuito. Por otra parte, diseñar los ECC para cubrir los casos comunes, pero obviando el peor caso, reducirá el retardo y el consumo, pero un cambio en el entorno que provoque una variación en la tasa de fallos provocará un mal comportamiento del ECC, ya que éste no se podrá adaptar a este cambio [11][12]. Por ejemplo, un aumento en la intensidad de la radiación cósmica que recibe un satélite o una sonda espacial debido al incremento de la actividad solar causaría una variación en la tasa de fallos.

Si bien la idea de la tolerancia a fallos adaptativa no es nueva [13], en la actualidad está ganando importancia, especialmente la adaptación basada en ECC, pues permite acelerar el cambio en las condiciones de tolerancia a fallos [14][15][16][17][18].

En este trabajo presentamos un mecanismo de tolerancia a fallos basado en un único ECC que puede cambiar su cobertura de fallos en función de los cambios que se produzcan en las condiciones de error. Las principales

Proyecto PID2020-120271RB-I00 financiado por MCIN/AEI /10.13039/501100011033

Todos los autores están adscritos al Instituto ITACA, Universitat Politècnica de València, Camino de Vera s/n, 46022 Valencia, España.

E-mail: { jgracia, ljsaiz, jcbaraza, dgil, pgil }@itaca.upv.es

Dirección de correo: Escuela Técnica Superior de Ingeniería Informática, Edificio 1G, Despacho 2S7, Universitat Politècnica de València, Camino de Vera s/n, 46022 Valencia, España.

Autor de contacto: J. Gracia Morán

novedades de este nuevo mecanismo de tolerancia a fallos son:

- El ECC tiene un único codificador. Una de las claves de esta propuesta es que la matriz de paridad es única para las distintas coberturas, por lo que el codificador es el mismo para todas ellas. Esto implica también que el número de bits redundantes es el mismo, sea cual sea la cobertura de fallos activa. De esta forma, no es necesaria la recodificación de los datos en memoria cuando las condiciones de error cambien. En otras propuestas se presentan diferentes aproximaciones. En [14] se utilizan diferentes decodificadores en los que, pese a tener la misma redundancia, es necesario recodificar toda la información almacenada en memoria cuando varía la tasa de fallos, con el consiguiente consumo de energía y pérdida de prestaciones. En cambio, en [15], se necesita un costoso procesamiento previo para identificar qué decodificador puede corregir el error producido. En este caso también se mantiene el mismo número de bits de paridad. Otro ejemplo sería el presentado en [16], en el que se usan diferentes codificadores y decodificadores, con diferentes propiedades de tolerancia a fallos y distinto número de bits de paridad.
- Número de decodificadores variable. Otra de las claves para entender nuestra propuesta es el hecho de que, aunque una matriz de paridad permita una cobertura de errores determinada, no es necesario que el decodificador la implemente completamente. Con la misma matriz de paridad, un decodificador con una cobertura menor será más rápido y consumirá menos. Así, se pueden emplear distintos decodificadores, con diferentes propiedades de tolerancia a fallos, si es necesario.

Este trabajo se organiza de la siguiente manera. La Sección II presenta una serie de trabajos previos en tolerancia a fallos adaptativa. La Sección III presenta los errores más comunes que se producen en memoria, una breve introducción al funcionamiento de los códigos de corrección de errores y el diseño del ECC adaptativo. La Sección IV describe diferentes configuraciones posibles del Mecanismo de Tolerancia a Fallos Adaptativo y los resultados de su síntesis en FPGA. Por último, la Sección V concluye este trabajo.

II. Trabajos previos de Tolerancia a Fallos Adaptativa

Un sistema que sea confiable cuando se enfrenta a cambios en el entorno se denomina resiliente. En este sentido, un sistema informático resiliente es un sistema capaz de adaptar, en tiempo de ejecución, sus mecanismos de tolerancia a fallos para cumplir con sus requisitos de tolerancia a fallos [19]. Es decir, el sistema debe adaptarse a los cambios del entorno para seguir funcionando tal y como se espera de él.

Una opción para implementar sistemas resilientes frente a fallos hardware es la combinación de diferentes mecanismos que puedan tolerar este tipo de fallos. Por ejemplo, en [20] se cambia entre varios códigos de Hamming (con diferente redundancia) según la acumulación de fallos existente. Por otra parte, [21] propone la combinación de códigos de Hamming y votadores para la corrección de errores en memorias de tipo *Resistive RAM*.

Otra opción utilizada en la implementación de sistemas adaptativos es el uso de circuitos programables del tipo FPGA (del inglés *Field Programmable Gate Arrays*). En un principio, la adaptabilidad se puede implementar mediante el uso de recursos libres de la FPGA, o reprogramando (total o parcialmente) el dispositivo [14]. El principal problema de estos métodos es que, o se mantiene parte de la FPGA sin usar, o se debe dedicar tiempo extra por si hay que reprogramar el dispositivo. El tiempo de reprogramación se puede disminuir realizando una reprogramación parcial, la cual permite reprogramar una parte de la FPGA [14][22].

Otras posibilidades utilizando FPGA son los diseños redundantes [14][22], donde solo se activa una parte del sistema, con lo que algunos de los recursos no se utilizan hasta que aumenta la tasa de fallos, desperdiciándolos; o mediante el uso de una combinación de diferentes técnicas de tolerancia a fallos [23].

La idea de utilizar diferentes mecanismos de tolerancia a fallos en un sistema y emplear solo uno de ellos es relativamente común. Por ejemplo, [24] propone un método adaptativo para tolerar fallos en memorias que implementa tres tipos diferentes de ECC, como son el bit de paridad, un código de Hamming y un código LPC (*Linear Product Code*). El hecho de tener diferentes tipos de ECC implica, por un lado, que cada ECC tiene un número diferente de bits de paridad, y, por otro lado, que hay que recodificar los datos cada vez que se cambia de ECC.

Otra de las opciones es la definición de diferentes niveles de error, y definir para cada nivel el mecanismo de tolerancia a fallos más adecuado. Por ejemplo, [25] define cuatro niveles de error con el fin de corregir fallos múltiples adyacentes. En el caso de [26], y utilizando códigos Reed-Solomon, se definen también diferentes niveles de protección. Otro ejemplo similar se puede ver en [27], en el que también se definen varios niveles de error, y a cada nivel se aplica un ECC distinto. En estos dos últimos ejemplos, el número de bits de paridad varía a medida que se necesita una mayor tolerancia a fallos, lo que implica que cuando se detecta un error, se tiene que recodificar el bloque de datos. Esta misma aproximación se presenta en [28]. En este caso, los autores utilizaron un código BCH (Bose–Chaudhuri–Hocquenghem), dividieron la memoria en bloques y, en función de un mapa de bloques de fiabilidad calculado previamente, asignan un ECC diferente a cada bloque. Cada ECC tiene un número diferente de bits de código, dependiendo de si el bloque necesita una mayor o menor protección contra errores. De hecho, usan el mismo código BCH habilitando o deshabilitando algunos módulos del codificador y decodificador según las necesidades de confiabilidad del bloque.

El código BCH también se utiliza en [29]. En este caso, los autores, primero, identifican el tipo de error producido, y después, este es corregido por el módulo corrector de errores adecuado. La idea es intentar obtener una alta eficiencia de decodificación y un bajo consumo de energía. Este trabajo también emplea un codificador único, que divide el decodificador en dos correctores (uno para errores simples y otro para errores dobles). Esta misma aproximación de utilizar

un mismo código BCH, pero activando o desactivando diferentes capacidades de tolerancia a fallos se emplea en [30] o en [31]. En ambos casos, el objetivo principal es minimizar la energía consumida por el ECC.

Otra aproximación se puede ver en [32] o en [33]. En [32], y para datos críticos, agrupa varios bloques de datos codificados. De esta manera, aumenta tanto el número de bits de datos como el número de bits de paridad, permitiendo aumentar de esta forma la tolerancia a fallos, pero a costa de incrementar la latencia de lectura por el proceso de codificación, el cual es más complejo. En el caso de [33], y en función de la tasa de errores, se divide el bloque de datos en dos partes, y añadiendo ceros en posiciones conocidas, recodifica los datos, aumentando de esta forma la tolerancia a fallos.

A alto nivel, la tolerancia a fallos adaptativa se puede implementar mediante cambios de software o de hardware. Por ejemplo, en [34], por un lado, mediante software, se activan o desactivan diferentes mecanismos de tolerancia a fallos implementados por software, y por otro lado, se genera un conjunto de tareas para soportar estos diferentes mecanismos de tolerancia a fallos. En [35] se protege la cache, cambiando en tiempo de ejecución, entre un sistema sin protección en este elemento a un sistema con protección. Otra opción se presenta en [36], donde también se cambia en tiempo de ejecución de un sistema con dos núcleos que ejecutan tareas de forma independiente, a un sistema que junta varios núcleos para formar un núcleo lógico que permite ejecutar la misma tarea de forma replicada.

III. Introducción al Diseño de Códigos Correctores de Errores

A. Tipos comunes de errores en memoria

En función del número de bits afectados, los errores en memoria se pueden catalogar como *errores simples* o *errores múltiples*. Un *error simple*, por ejemplo, ocasionado por el impacto de una partícula de radiación cósmica, afecta a un único bit. Se produce por lo que en inglés se conoce como *Single Event Upsets* (SEU) [37].

Como ya se ha comentado, el continuo aumento de la escala de integración de los circuitos CMOS ocasiona una mayor frecuencia de aparición de *errores múltiples* [2][3]. Estos errores se pueden clasificar en errores *aleatorios*, *adyacentes* o *de ráfaga*. Los *errores múltiples aleatorios* afectan a varios bits repartidos de forma aleatoria dentro de una palabra de datos. Los *errores múltiples adyacentes* afectan a bits contiguos, mientras que los *errores de ráfaga* afectan a varios bits contiguos, pero solo se sabe que el primer y el último bit de la ráfaga son erróneos; del resto de bits intermedios no se sabe qué ha ocurrido. La Fig. 1 muestra diferentes ejemplos de errores en memoria.

B. Conceptos básicos de Códigos Correctores de Errores

En un Código Corrector de Errores (ECC) binario (n, k), una palabra de entrada de k bits se codifica en una palabra de salida de n bits [5]. La palabra de entrada se puede representar mediante un vector de k bits con la forma $\mathbf{u}=(u_0, u_1, \ldots, u_{k-1})$, mientras que la palabra codificada se puede representar como un vector de n bits $\mathbf{b}=(b_0, b_1, \ldots, b_{n-1})$ donde los $(n-k)$ bits agregados se denominan bits de paridad, bits de código o bits redundantes.

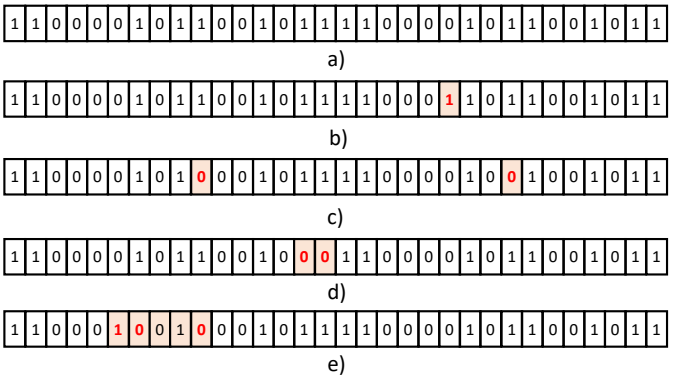


Fig. 1. Ejemplos de errores en memoria: a) dato original; b) error simple; c) error doble aleatorio; d) error doble adyacente; e) error de ráfaga de 5 bits.

Cuando el vector $\mathbf{b}$ se transmite a través de un canal no confiable, la palabra recibida $\mathbf{r}=(r_0, r_1, \ldots, r_{n-1})$ puede verse afectada por errores. Los errores inducidos por el canal se modelan mediante el vector de error $\mathbf{e}=(e_0, e_1, \ldots, e_{n-1})$. Si ha ocurrido un error en el i-ésimo bit, entonces $e_i=1$; de lo contrario, $e_i=0$. Así, se puede interpretar $\mathbf{r}$ como $\mathbf{r} = \mathbf{b} \oplus \mathbf{e}$.

Un código de bloque lineal se define mediante una matriz de comprobación de paridad $\mathbf{H}_{(n-k)\times n}$ [5]. De esta forma, durante el proceso de codificación, $\mathbf{b}$ debe cumplir $\mathbf{H}\cdot\mathbf{b}^T=\mathbf{0}$. El síndrome $\mathbf{s}$, utilizado en la decodificación, se define como $\mathbf{s}^T=\mathbf{H}\cdot\mathbf{r}^T$, y depende exclusivamente de $\mathbf{e}$, tal y como se muestra en (1). De esta manera, para cada error $\mathbf{e}$ corregible, debe haber un síndrome $\mathbf{s}$ diferente. Por lo tanto, podemos suponer que $\mathbf{e}=\mathbf{0}$ si $\mathbf{s}=\mathbf{0}$. En este caso, $\mathbf{r}$ es correcto. En caso contrario, se ha producido un error.

$$\mathbf{s}^T = \mathbf{H}\cdot\mathbf{r}^T = \mathbf{H}\cdot(\mathbf{b} \oplus \mathbf{e})^T = \mathbf{H}\cdot\mathbf{b}^T \oplus \mathbf{H}\cdot\mathbf{e}^T = \mathbf{H}\cdot\mathbf{e}^T \qquad (1)$$

Para ejecutar la decodificación basada en síndromes, cada vector de error estimado $\hat{\mathbf{e}}$ se relaciona con cada síndrome $\mathbf{s}$ mediante una tabla de búsqueda. La palabra decodificada $\hat{\mathbf{b}}$ se calcula como $\hat{\mathbf{b}} = \mathbf{r} \oplus \hat{\mathbf{e}}$. A partir de $\hat{\mathbf{b}}$, es fácil obtener el vector de datos decodificados $\hat{\mathbf{u}}$, simplemente descartando los bits de paridad, ya que el código es separable. Los datos decodificados $\hat{\mathbf{u}}$ deben ser iguales a los originales $\mathbf{u}$ con una probabilidad muy alta, si las hipótesis de fallos empleadas para diseñar el ECC son consistentes con el comportamiento del canal. La Fig. 2 sintetiza este proceso de codificación, cruce del canal y decodificación.

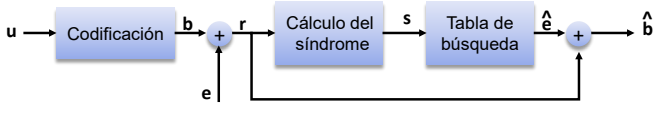


Fig. 2. Codificación, cruce del canal y proceso de decodificación.

C. Diseño del Código Corrector de Errores adaptativo

Tal y como se acaba de comentar, podemos describir un

ECC mediante su matriz de paridad **H**. Sin embargo, la búsqueda de una matriz **H** que cumpla con unos determinados requisitos de tolerancia a fallos puede ser muy compleja. Para encontrar la matriz de paridad que define el ECC que se presenta en este trabajo, hemos utilizado una metodología de búsqueda basada en patrones de error a corregir y/o detectar [38]. Estos patrones de error se describen utilizando un conjunto dado de vectores de error. Aunque esta metodología se empleó inicialmente para diseñar ECC con distintas coberturas de fallos dentro de la misma palabra de datos (códigos FUEC [38]), también se ha utilizado con éxito para crear diferentes familias de códigos (por ejemplo, [8][10][39] [40][41]). La Fig. 3 muestra un resumen de la metodología utilizada, describiéndose también brevemente a continuación.

Después de determinar el número de bits de la palabra de datos (k) y la palabra de código (n) para el ECC a diseñar, se seleccionan los patrones de error a corregir y/o detectar. A continuación, se busca la matriz de paridad **H**, que satisface las ecuaciones (2) y (3), donde $\boldsymbol{E}_+$ representa el conjunto de vectores de error a corregir, y $\boldsymbol{E}_\Delta$ el conjunto de vectores de error a detectar.

$$\boldsymbol{H} \cdot \boldsymbol{e}_i^T \neq \boldsymbol{H} \cdot \boldsymbol{e}_j^T; \forall \boldsymbol{e}_i, \boldsymbol{e}_j \in E_+ | \boldsymbol{e}_i \neq \boldsymbol{e}_j \quad (2)$$

$$\boldsymbol{H} \cdot \boldsymbol{e}_i^T \neq \boldsymbol{H} \cdot \boldsymbol{e}_j^T; \forall \boldsymbol{e}_i \in E_\Delta, \boldsymbol{e}_j \in E_+ \quad (3)$$

Cada error corregible debe tener un síndrome diferente, y cada error detectable debe generar un síndrome diferente a todos los síndromes generados por los errores corregibles

En el caso de nuestra propuesta, hemos diseñado un ECC que, en su máxima capacidad de tolerancia a fallos, puede corregir errores simples y errores de ráfaga de 2 y 3 bits; además, puede detectar errores de ráfaga de 4 bits. De esta forma, $\boldsymbol{E}_+$ debe incluir los siguientes vectores de error:

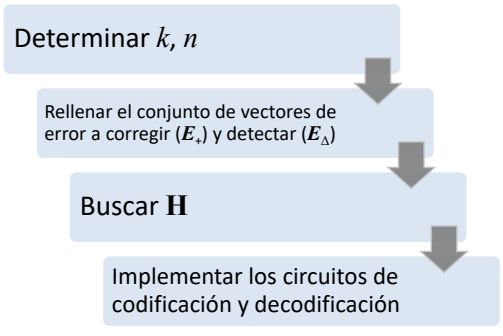
- Los errores individuales, que se representan con vectores (…1…).
- Los errores de ráfaga de 2 bits, que tienen vectores (…11…).
- Los vectores de error de ráfaga de 3 bits, representados mediante vectores del tipo (…1x1…).

Los puntos representan cero o más ceros, la x puede ser 0 o 1, y el 1 representa el error. De la misma manera, $\boldsymbol{E}_\Delta$ debe incluir vectores del tipo (…1xx1…), que representan los errores de ráfaga de 4 bits.

Para encontrar la matriz **H**, se utiliza el algoritmo de búsqueda recursivo mostrado en la Fig. 3b. Este algoritmo verifica matrices parciales y agrega una nueva columna solo si la matriz anterior satisface los requisitos. Las columnas agregadas deben ser distintas de cero, por lo que hay $2^{(n-k)} - 1$ combinaciones para cada columna. Para encontrar matrices con un peso de Hamming bajo, primero se comprueban las columnas con un número menor de unos ("1"). Un número bajo de "1" en la matriz servirá para construir circuitos de codificación y decodificación más rápidos, más pequeños y que consuman menos energía. Por otra parte, el algoritmo permite mejorar la generación de la matriz **H** para reducir el peso de Hamming en la fila más pesada, así como reducir el peso de Hamming de toda la matriz.

Una vez encontrada la matriz **H**, es fácil obtener las ecuaciones lógicas para los circuitos codificadores y el cálculo del síndrome en los circuitos decodificadores, como se puede ver en [38]. Posteriormente, es necesario relacionar cada síndrome con el vector de error correspondiente, implementando una tabla de búsqueda. De hecho, estas son las tablas de verdad de un grupo de funciones lógicas donde las entradas son los bits del síndrome y las salidas son los bits de los vectores de error estimados. Cada función lógica puede ser una suma de minitérminos que representan los síndromes que afectan a cada bit. En [39] se puede encontrar una buena explicación de cómo generar la tabla de búsqueda y las ecuaciones lógicas para los bits del vector de error estimado.

En [38] se muestra una explicación más detallada de la metodología, así como un ejemplo de diseño de un ECC.



(a)

```
Procedure CheckPartialMatrix partial_H (n-k) × ncols  /* ncols ∈ [1..n] */
    SyndromeSet = {}
    For each error vector e in E+
        partial_e = (e0, e1, ..., encols-1)
        If HammingWeight(e) = HammingWeight(partial_e)
            newSyndrome = CalculateSyndrome(partial_H × Transpose(partial_e))
            If newSyndrome in SyndromeSet then Return    /* Wrong partial matrix */
            Else Add newSyndrome to SyndromeSet
        End if
    End for
    For each error vector e in EΔ
        partial_e = (e0, e1, ..., encols-1)
        If HammingWeight(e) = HammingWeight(partial_e)
            newSyndrome = CalculateSyndrome(partial_H × Transpose(partial_e))
            If newSyndrome in SyndromeSet then Return    /* Wrong partial matrix */
            Else Do Nothing   /* newSyndrome not stored in this case */
        End if
    End for
    If ncols = n
        Add partial_H to SolutionsSet
        Return
    Else
        For each possible new_column    /* n- k bits, excluding the all 0 combination */
            CheckPartialMatrix[partial_H | new_column] (n- k) × (ncols+1)
        End for
    End if
End procedure
```

(b)

Fig. 3. Metodología de búsqueda de matrices de paridad basada en el conjunto de patrones de error a corregir y/o detectar: (a) Diagrama de flujo; (b) Algoritmo de búsqueda (extraído de [38]).

IV. Propuesta de Mecanismo de Tolerancia a Fallos Adaptativo

En esta sección se describe el sistema utilizado para comprobar el comportamiento del mecanismo de tolerancia a fallos adaptativo, así como los resultados que se han obtenido al sintetizar este sistema en diferentes FPGA.

A. Definición del sistema

Para comprobar la viabilidad de nuestra propuesta, hemos utilizado el modelo en VHDL del procesador Microsimplez [42]. Es un modelo de procesador teórico con arquitectura Von Neumann, palabras de datos de 16 bits y una memoria RAM de 512 palabras. Este sencillo procesador lo hemos utilizado para tener una base común en la que añadir diferentes ECC y poder estudiar su impacto en las diferentes sobrecargas que provoca esta inclusión. Como carga de trabajo, hemos utilizado el cálculo de la serie de Fibonacci.

En cuanto al ECC, se propone como ejemplo uno adecuado para la memoria del procesador Microsimplez. No obstante, la técnica utilizada aquí se puede aplicar a cualquier otro código. En la Fig. 4 se puede ver la matriz de paridad **H**, para los requisitos especificados, y obtenida mediante el algoritmo del apartado anterior. Esta matriz permite diseñar un código que puede corregir errores simples y errores de ráfaga de 2 y 3 bits, así como detectar errores de ráfaga de 4 bits. Nuestra propuesta consiste en definir diferentes decodificadores con diversas capacidades de corrección y/o detección, reduciendo esta capacidad máxima con el objetivo de obtener decodificadores más simples, rápidos y que consuman menos. En cualquier caso, el codificador siempre es el mismo para todos los decodificadores, así como la redundancia utilizada, lo que permite alternar entre los distintos decodificadores, dependiendo de las condiciones de trabajo.

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

Fig. 4. Matriz de paridad **H** del ECC adaptativo propuesto.

Si se utilizasen distintos códigos para las diferentes coberturas (por ejemplo, un código Hamming para errores simples y un BCH para errores dobles), sería necesario disponer de los dos circuitos codificadores, y recodificar la información al cambiar de uno a otro. Utilizar un codificador único permite, por un lado, ahorrar área de silicio y consumo de potencia, y, por otro lado, no es necesario recodificar la información para aplicar las distintas coberturas de los diferentes decodificadores. De esta forma, es muy fácil añadir tantos decodificadores como sea necesario.

En cuanto al número de bits redundantes, nuestra propuesta utiliza 8 bits de código para 16 bits de datos, formando un ECC (24, 16). Si tenemos en cuenta que sólo se utiliza un codificador, todos los decodificadores que se implementen utilizarán el mismo número de bits redundantes. En general, las coberturas más bajas se podrían obtener utilizando menos bits de código, pero eso supondría complicar la organización de un esquema adaptativo como el propuesto, además de traducirse en decodificadores más complejos. En nuestra propuesta, al mantener la redundancia y reducir la cobertura, se obtienen circuitos decodificadores más rápidos y con un consumo menor.

Otro aspecto a tener en cuenta es que los trabajos comentados en la Sección II utilizan como códigos de corrección de errores OLS o BCH. Si bien estos dos ECC son capaces de corregir errores aleatorios, la corrección de este tipo de errores es más compleja que la corrección de errores adyacentes, principalmente porque existen más vectores de error. Así, estos dos códigos son más complejos de implementar y añadirían más sobrecargas que nuestra propuesta.

En este trabajo, y a modo de ejemplo, hemos definido tres decodificadores o niveles de error, los cuales permiten mostrar la versatilidad y adaptabilidad de nuestra propuesta. Obviamente, se podrían definir más (o menos) niveles de error, y otros tipos de coberturas, de acuerdo con las necesidades del sistema. Los niveles de error definidos son:

- Corrección de errores de un solo bit y detección de errores de ráfaga de 2 bits (al que hemos denominado SEC-2bBED, del inglés *Single Error Correction – 2-bit Burst Error Detection*);
- Corrección de errores de un bit y de ráfagas de 2 bits y detección de errores en ráfagas de 3 bits (al que hemos llamado 2bBEC-3bBED, del inglés *2-bit Burst Error Correction – 3-bit Burst Error Detection*);
- Corrección de errores simples y de ráfagas de 2 y 3 bits, y detección de errores en ráfagas de 4 bits (denominado en este caso 3bBEC-4bBED, del inglés *3-bit Burst Error Correction – 4-bit Burst Error Detection*).

B. Modelos VHDL utilizados

Como ejemplo de la funcionalidad del ECC adaptativo, hemos implementado diferentes modelos tolerantes a fallos de la CPU Microsimplez a partir de nuestra propuesta.

En el primer modelo, vamos a trabajar con una tasa de fallos fija en el espacio, esto es, vamos a suponer que todos los bits de la memoria tienen la misma probabilidad de sufrir fallos. La Fig. 5 muestra el diagrama de bloques de este modelo, donde el decodificador puede ser cualquiera de los comentados en el punto anterior.

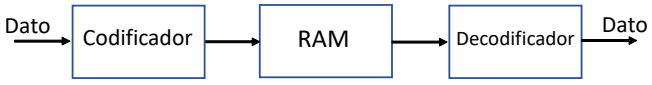


Fig. 5. Memoria RAM del la CPU Microsimplez con un ECC.

Para el segundo modelo de la CPU Microsimplez Tolerante a Fallos, se va a trabajar con una tasa de fallos variable en el espacio, es decir, hemos supuesto que unos bloques de memoria son más propensos a determinados fallos que otros. Con este objetivo, se han definido cuatro zonas diferentes en el módulo de la memoria RAM, tal y como se muestra en la Fig. 6. Para proteger las zonas 1 y 4, hemos utilizado el decodificador SEC-2bBED. Para proteger la zona 2, hemos

empleado el decodificador 2bBEC-3bBED. Y finalmente, para proteger la zona 3, se ha utilizado el decodificador 3bBEC-4bBED. Esta configuración está inspirada en trabajos como los presentados en [25][26][27][28], donde los autores dividen la memoria en bloques y, en función de un mapa de bloques de fiabilidad calculado previamente, asignan un ECC diferente a cada bloque. Por ejemplo, en [26] y [27] se cambia de ECC en función de la tasa de errores, mientras que en [28] cada ECC tiene un número diferente de bits de código, dependiendo de si el bloque necesita una mayor o menor protección contra errores. En nuestro ejemplo, todos los ECC tienen el mismo número de bits redundantes.

Obviamente, son posibles más configuraciones. Las dos que se acaban de mostrar son únicamente dos ejemplos factibles.

Por otra parte, también hemos sintetizado modelos con otros ECC para poder realizar una comparación más completa. En concreto, se ha sintetizado el sistema original no tolerante a fallos (denominado Microsimplez), y también se han diseñado y sintetizado otras propuestas que incluyen dos ECC que muestran una buena combinación entre su velocidad de decodificación y un bajo número de bits redundantes. Estos ECC son el BCH [43] y el FUEC-QUAEC [8]. En todos los ejemplos, se ha sintetizado el sistema completo, es decir, el bloque de CPU, el módulo de memoria RAM y los diferentes componentes de los ECC.

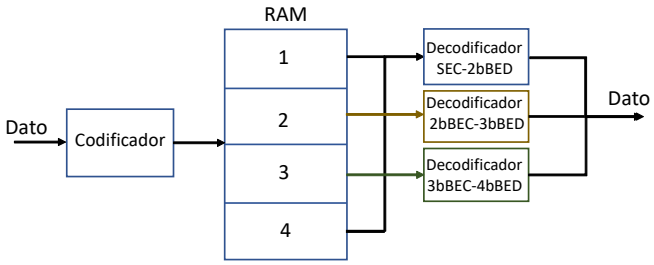


Fig. 6. CPU Microsimplez con el ECC Adaptativo.

C. Resultados de la síntesis en FPGA

Los diferentes modelos se han sintetizado en una placa de evaluación Zynq-7 ZC702, que permite una reprogramación parcial de la FPGA. Esta placa incorpora un SoC Xilinx XC7Z020-1CLG484C [44] y un microprocesador ARM® Cortex®-A9 MPCoreTM integrado, que se puede ocupar de la reprogramación lógica de la FPGA.

Un aspecto importante de un ECC es la redundancia que introduce. Esta afecta tanto al tamaño de la memoria como a la complejidad de los circuitos de codificación y decodificación, y, por lo tanto, al área de silicio y al consumo de potencia. Si tenemos en cuenta que el número de bits de datos es 16, la Tabla I muestra esta redundancia para cada ECC utilizado, calculada como:

$$Redundancia = \frac{Nº\ bits\ de\ código}{Nº\ bits\ de\ datos} \times 100 \qquad (4)$$

TABLA I
REDUNDANCIA DE LOS ECC UTILIZADOS

ECC	Nº bits código	Redundancia (%)
SEC-2bBED	8	50.00
2bBEC-3bBED	8	50.00
3bBEC-4bBED	8	50.00
BCH [43]	10	62.50
FUEC-QUAEC [8]	9	56.25
ECC Adaptativo	8	50.00

Como se puede ver, nuestra propuesta introduce el menor número de bits de código. Este hecho influye en la cantidad de bits de memoria adicionales necesarios, ya que cada palabra de memoria almacena bits de datos más bits de código.

Por otro lado, y con respecto a los circuitos de codificación y decodificación, su tamaño depende tanto de las capacidades de corrección y/o detección de errores como del número de bits de paridad. En el caso de nuestras propuestas, la complejidad introducida por los circuitos codificadores y decodificadores es compensada por su menor redundancia, lo que ocasiona sobrecargas aceptables.

Los resultados obtenidos en las diferentes síntesis de los distintos modelos comentados anteriormente se pueden ver en la Tabla II. Las primeras dos columnas de esta tabla muestran la velocidad máxima (y el mínimo período) alcanzados por cada uno de los modelos, mientras que en el resto de las columnas se pueden ver los resultados de la síntesis para la menor de esas frecuencias máximas.

Obviamente, el modelo más rápido es el modelo no tolerante a fallos. En cuanto a los modelos que integran algún ECC, el más rápido es nuestra propuesta SEC-2bBED. La lógica combinacional de este ECC es más simple que la del resto, y por ello, se ejecuta más rápidamente.

Por otra parte, el modelo más lento es el que incluye el BCH. Este ECC es capaz de corregir dos bits erróneos aleatorios dentro de la misma palabra, pero para ello su lógica combinacional es más compleja que la del resto, penalizando así su tiempo de ejecución.

La segunda parte de la Tabla II muestra el hardware utilizado y su consumo de energía para la frecuencia de funcionamiento de 27 MHz (la frecuencia máxima a la que funciona el modelo más lento). Como era de esperar, el modelo sin tolerancia a fallos es el que menos consumo de hardware y energía presenta.

Con respecto a los modelos tolerantes a fallos, los modelos con el BCH y el FUEC-QUAEC son los que ocupan más hardware y consumen más energía. Por un lado, estos ECC son los que presentan una mayor redundancia, lo que implica un tamaño de memoria mayor. Por otro lado, la complejidad de sus expresiones para calcular los bits de código (tanto en la codificación como en la decodificación) provocan este mayor consumo de hardware y energía. En concreto, el consumo estático es lo que consume la FPGA solo por estar encendida. Al utilizar más hardware (LUTs y FFs), el consumo estático de estos dos modelos es el mayor. Por otra parte, al ser la implementación de estos circuitos no muy grande, el consumo estático domina al consumo dinámico.

TABLA II
RESULTADOS DE LA SÍNTESIS EN ZYNQ-7 ZC702 (CPU + RAM + ECC)

	Velocidad Máxima		Frecuencia funcionamiento = 27 MHz				
			Hardware usado		Consumo de Energía (W)		
	f_{max} (MHz)	T_{min} (ns)	LUT	FF	Estático	Dinámico	Total
µSimplez	56	18	2433	170	0.044	0.105	0.149
SEC-2bBED	36	28	4199	225	0.104	0.106	0.210
2bBEC-3bBED	33	30	4227	225	0.102	0.106	0.208
3bBEC-4bBED	31	32	4261	227	0.101	0.106	0.207
BCH [43]	27	37	4883	261	0.140	0.106	0.246
FUEC-QUAEC [8]	32	31	4512	245	0.111	0.106	0.217
ECC Adaptivo	31	32	4281	225	0.107	0.106	0.213

En cuanto a nuestra propuesta, podemos ver que el ECC Adaptativo tiene un consumo de hardware y de energía ligeramente mayor que el modelo con el ECC denominado 3bBEC-4bBED. Aunque es un resultado esperado, ya que se debe disponer de los tres decodificadores para utilizar uno u otro según convenga, evoluciones futuras de este esquema podrían disponer de mecanismos de desconexión física de los decodificadores no usados para reducir el consumo. En todo caso, este modelo ocupa menos área de silicio y consume menos energía que los modelos con el BCH y el FUEC-QUAEC. Es decir, nuestras propuestas adecúan la tolerancia a fallos sin aumentar la sobrecarga con respecto al consumo de hardware o energía.

Un aspecto que debemos remarcar es que el procesador utilizado (CPU Microsimplez) es una CPU teórica. Con la implementación en la FPGA utilizada, la frecuencia de reloj debe ser la del caso más lento, el de mayor cobertura. Con un procesador más real, y con otra FPGA más adecuada al contexto, que permitiese que la frecuencia varíe en tiempo de ejecución, sería posible incrementar las frecuencias de funcionamiento de los decodificadores más simples.

La gran ventaja de la placa de evaluación Zynq-7 ZC702 es que permite realizar una reprogramación parcial, es decir, es posible cambiar el decodificador a medida que la tasa de fallos varía. Esto permite proponer un nuevo sistema adaptativo, en el que el ECC aplicado dependiera de la tasa de errores, y pudiera cambiar este ECC de manera dinámica.

La idea de funcionamiento sería la siguiente. El sistema empezaría a funcionar con el decodificador más sencillo (el SEC-2bBED). Cuando la tasa de fallos aumente, este decodificador se cambiaría por el siguiente (2bBEC-3bBED). Este cambio puede hacerse, por ejemplo, fijando un umbral de número de fallos que no se hayan podido corregir [14]. Si la tasa de fallos sigue aumentando, se podría realizar un último cambio de decodificador, y así utilizar el denominado 3bBEC-4bBED. Obviamente, si la tasa de fallos disminuye, también es posible utilizar un decodificador con menos capacidad de tolerancia a fallos.

En estos casos, un aspecto importante que hay que tener en cuenta es el tiempo de reprogramación, que depende de la tecnología de la FPGA (es decir, del tiempo de lectura del puerto de acceso a la configuración del procesador o PCAP – *Processor Configuration Access Port* – utilizado para reprogramar la FPGA), del tiempo de lectura de la "ROM de configuración" y del tamaño del fichero *bitstream* que se va a utilizar en la configuración. Este tiempo se puede estimar con la siguiente expresión [14]:

$$t_{reconf\_module} = \frac{M}{ROM_{read\_rate}} + t_{PCAP\_init} + \frac{M}{PCAP_{read\_rate}}$$

donde M es el tamaño del fichero *bitstream* (en megabytes), ROM_{read_rate} es el ancho de banda de lectura de la ROM (en megabytes por segundo); t_{PCAP_init} es un retardo de inicialización fijo del puerto (en segundos); y $PCAP_{read\_rate}$ se corresponde con el ancho de banda de lectura del puerto PCAP (en megabytes por segundo). A partir de [14] y [44], podemos obtener los valores para la FPGA XC7Z020, que son: $t_{PCAP\_init} = 0.0005\ s$ y $PCAP_{read\_rate} = 120\ MBytes/s$. La sincronización de la tarjeta SD utilizada como "ROM de configuración" es $ROM_{read\_rate} = 19.63\ MBytes/s$. Y, por último, el tamaño de todos los archivos *bitstream* es de 201 KBytes. De esta forma, $t_{reconf\_module} = 12.135\ ms$.

La Tabla III muestra los resultados obtenidos en la síntesis de este modelo, teniendo en cuenta las características de reprogramación parcial de la FPGA utilizada. Hay que indicar que estos resultados están marcados por el microprocesador ARM de la placa, el cual es necesario para realizar la reprogramación parcial del sistema. En este sentido, incluso la frecuencia máxima de funcionamiento del sistema viene marcada por el ARM. En concreto, se puede ver que la frecuencia máxima de funcionamiento es algo menor que la del ECC Adaptativo sin reprogramación parcial visto anteriormente y mostrado en la Tabla II. En cuanto a las LUT utilizadas, se obtiene un valor algo mayor que el obtenido para el SEC-2bBED, y un número de FF superior a casi todos los ECC. Esto es debido a que, aunque se tiene sólo un decodificador implementado, también está sintetizada la circuitería necesaria para la identificación adicional requerida en la reprogramación parcial. Por último, se puede ver que el consumo de energía es mucho mayor que en el resto de los modelos. Este mayor consumo es el que necesita el microprocesador ARM, del cual depende la reprogramación parcial del sistema, tal y como se ha comentado antes.

Hay que tener en cuenta que, aunque los valores que se muestran en esta sección no son buenos para un sistema crítico

en tiempo real, muestran la viabilidad de nuestro enfoque. También es importante señalar que esta FPGA se ha utilizado solo para probar la factibilidad del diseño propuesto. La búsqueda de la FPGA más adecuada para tal fin está fuera del alcance de este trabajo.

TABLA III
RESULTADOS DE LA SÍNTESIS EN ZYNQ-7 ZC702 (CPU + RAM + ECC) DEL MODELO "ECC ADAPTIVO"

Velocidad Máxima		
f_{max} (MHz)		**T_{min} (ns)**
29		35
Hardware usado		
LUT		**FF**
4217		258
Consumo de Energía (W)		
Estático	**Dinámico**	**Total**
0.140	1.627	1.767

V. Conclusiones

En este trabajo hemos presentado dos configuraciones de un mecanismo de tolerancia a fallos basado en un ECC adaptativo cuya principal característica es su único codificador. De esta forma, se puede utilizar un número variable de decodificadores diferentes en función de las necesidades de tolerancia a fallos del sistema. Además, todos los decodificadores presentan el mismo número de bits redundantes, lo que evita recodificar los datos al cambiar de decodificador.

Para probar la viabilidad de este nuevo mecanismo de tolerancia a fallos adaptativo, lo hemos incluido en una CPU muy simple y lo hemos sintetizado en una FPGA, activando y desactivando la característica de reprogramación parcial. Aún a pesar de utilizar una CPU muy sencilla, se pueden ver algunas de las ventajas de nuestra propuesta.

En primer lugar, se ha visto que la sobrecarga en el retardo de procesamiento de nuestras propuestas es similar al de otras soluciones no adaptativas. Nuestra propuesta añade su comportamiento adaptativo sin penalizar al sistema.

También se han observado diferentes consumos de energía de los distintos decodificadores implementados, en función de la tolerancia a fallos activa. Una menor capacidad de corrección/detección implica un menor consumo de energía. En este sentido, es posible disminuir este consumo cuando disminuye la tasa de fallos, pues se puede utilizar el decodificador con menor tolerancia a fallos.

Esta menor tolerancia a fallos también implica un menor retardo. De esta forma, en entornos en los que los no haya muchos fallos, se puede utilizar el decodificador más rápido (y que consume menos energía), a diferencia de los ECC no adaptativos, en los que el consumo de energía y el retardo siempre es el mayor.

Por último, en cuanto a los recursos hardware utilizados, sigue la misma tendencia que el consumo de energía. Es decir, una menor tolerancia a fallos implica una menor sobrecarga en el hardware.

En el futuro, queremos seguir desarrollando ECC con un solo codificador y diferentes decodificadores. Con respecto al enfoque presentado aquí, queremos probarlo con modelos de microprocesadores que modelen su comportamiento de forma más precisa, así como verificar su funcionamiento con más niveles de error y más configuraciones. Además, también queremos realizar un estudio más detallado de la sobrecarga temporal y las frecuencias máximas que podría alcanzar nuestra propuesta en modelos más precisos de microprocesadores, además de realizar síntesis en diferentes modelos de FPGA, con y sin reprogramación parcial.

Referencias

[1] IEEE International Roadmap for Devices and Systems 2030. Acceso 2 de febrero, 2023. [Online]. Disponible en: https://irds.ieee.org/editions/2023

[2] B.L. Bhuva et al., "Multi-cell soft errors at advanced technology nodes", IEEE Transactions on Nuclear Science, 62(6):2585–2591, 2015, doi: 10.1109/TNS.2015.2488630

[3] G. Tsiligiannis et. al., "Multiple Cell Upset Classification in Commercial SRAMs", IEEE Transactions on Nuclear Science, vol. 61, no. 4, August 2014, doi: 10.1109/TNS.2014.2313742

[4] N.G. Chechenin and M. Sajid, "Multiple cell upsets rate estimation for 65 nm SRAM bit-cell in space radiation environment", 3rd International Conference and Exhibition on Satellite & Space Missions, May 2017, doi: 10.4172/2168-9792-C1-017

[5] I. Koren and C.M. Krishna, Fault-Tolerant Systems, 2nd Edition, Morgan Kaufmann, 2021, doi: https://doi.org/10.1016/C2018-0-02160-X

[6] R. W. Hamming, "Error detecting and error correcting codes," Bell System Technical Journal, vol. 29, pp. 147–160, 1950, doi: 10.1002/j.1538-7305.1950.tb00463.x

[7] C.L. Chen and M.Y. Hsiao, "Error-correcting codes for semiconductor memory applications: a state-of-the-art review", IBM Journal of Research and Development, vol. 58, no. 2, pp. 124–134, March 1984, doi: 10.1147/rd.282.0124

[8] J. Gracia-Moran, L.J. Saiz-Adalid, D. Gil-Tomás, and P.J. Gil-Vicente, "Improving Error Correction Codes for Multiple Cell Upsets in Space Applications", IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 26(10), pp. 2132-2142, October 2018, doi: 10.1109/TVLSI.2018.2837220

[9] F. Garcia-Herrero, A. Sánchez-Macián, and J.A. Maestro, "Low delay non-binary error correction codes based on Orthogonal Latin Squares", Integration, vol. 76, pp. 55-60, January 2021, doi: https://doi.org/10.1016/j.vlsi.2020.09.004

[10] L.-J. Saiz-Adalid, J. Gracia-Morán, D. Gil-Tomás, J.-C. Baraza-Calvo, P.-J. Gil-Vicente, "Reducing the Overhead of BCH Codes: New Double Error Correction Codes", Electronics 2020, 9, 1897, doi: https://doi.org/10.3390/electronics9111897

[11] Y. Xu, I. Koren and C.M. Krishna, "AdaFT: A Framework for Adaptive Fault Tolerance for Cyber-Physical Systems", ACM Transactions on Embedded Computing Systems, vol. 16, nº 3, article No.: 79, pp 1–25, 2017, doi: 10.1145/2980763

[12] S.K. Lu, H.P. Li and K. Miyase, "Adaptive ECC Techniques for Reliability and Yield Enhancement of Phase Change Memory", 2018 IEEE 24th International Symposium on On-Line Testing and Robust System Design (IOLTS), pp. 226-227, 2018, doi: 10.1109/IOLTS.2018.8474118

[13] K.H.K. Kim and T. F. Lawrence, "Adaptive Fault Tolerance: Issues and Approaches", Second IEEE Workshop on Future Trends of Distributed Computing Systems. pp. 38–46, 1990, doi: 10.1109/FTDCS.1990.138292

[14] J.-C. Baraza-Calvo, J. Gracia-Morán, L.-J. Saiz-Adalid, D. Gil-Tomás, and P.-J. Gil-Vicente, "Proposal of an Adaptive Fault Tolerance Mechanism to Tolerate Intermittent Faults in RAM", Electronics, vol. 9, no. 12, p. 2074, Dec. 2020, doi: https://doi.org/10.3390/electronics9122074

[15] F. Silva, A. Muniz, J. Silveira, and C Marcon, "CLC-A: An Adaptive Implementation of the Column Line Code (CLC) ECC", 2020 33rd Symposium on Integrated Circuits and Systems Design (SBCCI), August 2020, doi: 10.1109/SBCCI50935.2020.9189901

[16] M. Hadizadeh, E. Cheshmikhani and H. Asadi, "STAIR: High Reliable STT-MRAM Aware Multi-Level I/O Cache Architecture by Adaptive

ECC Allocation", 2020 Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 1484-1489, 2020, doi: 10.23919/DATE48585.2020.9116550

[17] T. Zhan, X. Wang, D. Feng and W. Tong, "AetEC: Adaptive error-tolerant Erasure Coding Scheme Within SSDs", 2020 IEEE 38th International Conference on Computer Design (ICCD), pp. 167-174, 2020, doi: 10.1109/ICCD50377.2020.00041

[18] Y. S. Lee, G. Koo, Y. -H. Gong and S. W. Chung, "Stealth ECC: A Data-Width Aware Adaptive ECC Scheme for DRAM Error Resilience", 2022 Design, Automation & Test in Europe Conference & Exhibition (DATE), 2022, pp. 382-387, doi: 10.23919/DATE54114.2022.9774775

[19] W. Excoffon, J.C. Fabre, and M. Lauer, "Analysis of Adaptive Fault Tolerance for Resilient Computing", 2017 13th European Dependable Computing Conference (EDCC 2017), pp. 50-57, September 2017, doi: 10.1109/EDCC.2017.22

[20] S. Hemaram, M. Mayahinia and M. B. Tahoori, "Adaptive Block Error Correction for Memristive Crossbars", 2022 IEEE 28th International Symposium on On-Line Testing and Robust System Design (IOLTS), 2022, doi: 10.1109/IOLTS56730.2022.9897817

[21] A. Das and N. A. Touba, "Selective Checksum based On-line Error Correction for RRAM based Matrix Operations", 2020 IEEE 38th VLSI Test Symposium (VTS), 2020, doi: 10.1109/VTS48691.2020.9107606

[22] G. I. Alkady et al., "An adaptive multi-factor fault-tolerance selection scheme for FPGAs in space applications", 2018 7th Mediterranean Conference on Embedded Computing (MECO), Budva, Montenegro, 2018, doi: 10.1109/MECO.2018.8406019

[23] B. Shashidhara, S. Jadhav and Y. S. Kim, "Reconfigurable Fault Tolerant Processor on a SRAM based FPGA", 2020 IEEE International Conference on Electro Information Technology (EIT), pp. 151-154, 2020, doi: 10.1109/EIT48999.2020.9208275

[24] M. Stefani, C. Marcon, F. Silva and J. Silveira, "Memory Controller with Adaptive ECC for Reliable System Operation", 2023 36th SBC/SBMicro/IEEE/ACM Symposium on Integrated Circuits and Systems Design (SBCCI), 2023, doi: 10.1109/SBCCI60457.2023.10261959

[25] K. Wang, A. Louri, A. Karanth and R. Bunescu, "High-performance, energy-efficient, fault-tolerant network-on-chip design using reinforcement learning", 2019 Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 1166-1171, 2019, doi: 10.23919/DATE.2019.8714869

[26] T. Zhan, X. Wang, D. Feng and W. Tong, "AetEC: Adaptive error-tolerant Erasure Coding Scheme Within SSDs", 2020 IEEE 38th International Conference on Computer Design (ICCD), pp. 167-174, 2020, doi: 10.1109/ICCD50377.2020.00041

[27] S. -K. Lu, Z. -L. Tsai, C. -L. Hsu and C. -T. Sun, "Fault-Aware ECC Techniques for Reliability Enhancement of Flash Memory", 2020 International Symposium on VLSI Design, Automation and Test (VLSI-DAT), 2020, doi: 10.1109/VLSI-DAT49148.2020.9196286

[28] D. Shin, J. Park, J. Park, S. Paul and S. Bhunia, "Adaptive ECC for Tailored Protection of Nanoscale Memory", IEEE Design & Test, vol. 34, no. 6, pp. 84-93, Dec. 2017, doi: 10.1109/MDAT.2016.2615844

[29] S. Choi et al., "A Decoder for Short BCH Codes with High Decoding Efficiency and Low Power for Emerging Memories", IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 27, no. 2, pp. 387-397, February 2019, doi: 10.1109/TVLSI.2018.2877147

[30] S. Taneja and M. Alioto, "PUF Architecture with Run-Time Adaptation for Resilient and Energy-Efficient Key Generation via Sensor Fusion", in IEEE Journal of Solid-State Circuits, vol. 56, no. 7, pp. 2182-2192, July 2021, doi: 10.1109/JSSC.2021.3050959

[31] P. Rao, P. Babshet, R. Arun Babu and M. S. Sunita, "Encoder and Adaptive Decoder for a (15,6,2) DEC-TED BCH Code", 2020 IEEE 17th India Council International Conference (INDICON), 2020, doi: 10.1109/INDICON49873.2020.9342357

[32] D. Kim and J. Kim, "Adaptive Granularity On-die ECC", 2022 19th International SoC Design Conference (ISOCC), pp. 318-319, 2022, doi: 10.1109/ISOCC56007.2022.10031324

[33] M. Zhang et al., "ALCod: Adaptive LDPC Coding for 3-D NAND Flash Memory Using Inter-Layer RBER Variation," in IEEE Transactions on Consumer Electronics, vol. 69, no. 4, pp. 1068-1081, Nov. 2023, doi: 10.1109/TCE.2023.3319638

[34] J. Caplan, Z. Al-bayati, H. Zeng and B. H. Meyer, "Mapping and Scheduling Mixed-Criticality Systems with On-Demand Redundancy", in IEEE Transactions on Computers, vol. 67, no. 4, pp. 582-588, April 2018, doi: 10.1109/TC.2017.2762293

[35] F. Kempf, J. Hoefer, F. Kreß, T. Hotfilter, T. Harbaum and J. Becker, "Runtime Adaptive Cache Checkpointing for RISC Multi-Core Processors", 2022 IEEE 35th International System-on-Chip Conference (SOCC), 2022, doi: 10.1109/SOCC56010.2022.9908110

[36] F. Kempf and J. Becker, "Leveraging Adaptive Redundancy in Multi-Core Processors for Realizing Adaptive Fault Tolerance in Mixed-Criticality Systems", 2023 12th Mediterranean Conference on Embedded Computing (MECO), 2023, doi: 10.1109/MECO58584.2023.10154986

[37] K.A. LaBel, "Proton single event effects (SEE) guideline", Submitted for publication on the NASA Electronic Parts and Packaging (NEPP) Program website, August 2009. Disponible en https://nepp.nasa.gov/files/18365/Proton_RHAGuide_NASAAug09.pdf

[38] L.J. Saiz-Adalid et al., "Flexible unequal error control codes with selectable error detection and correction levels", 2013 Computer Safety, Reliability, and Security Conference (SAFECOMP 2013), pp. 178–189, September 2013, doi: https://doi.org/10.1007/978-3-642-40793-2_17

[39] L.J. Saiz-Adalid, J. Gracia-Morán, D. Gil-Tomás, J.C. Baraza-Calvo, P.J. Gil-Vicente, "Ultrafast Codes for Multiple Adjacent Error Correction and Double Error Detection", IEEE Access 2019, 7, 151131–151143, doi:10.1109/ACCESS.2019.2947315

[40] J. Gracia-Morán, L.J. Saiz-Adalid, J.C. Baraza-Calvo, D. Gil-Tomás, P.J. Gil-Vicente, "Design, Implementation and Evaluation of a Low Redundant Error Correction Code", IEEE Latin American Transactions, Vol. 19(11), pp. 1903 – 1911, November 2021, doi: 10.1109/TLA.2021.9475624

[41] J. Gracia-Moran, L.-J. Saiz-Adalid, J.-C. Baraza-Calvo and P. Gil, "Correction of Adjacent Errors with Low Redundant Matrix Error Correction Codes", 2018 Eighth Latin-American Symposium on Dependable Computing (LADC), pp. 107-114, 2018, doi: 10.1109/LADC.2018.00021

[42] https://opencores.org/projects/usimplez

[43] R. Naseer and J. Draper, "DEC ECC design to improve memory reliability in Sub-100nm technologies", 2008 15th IEEE Int. Conf. on Electronics, Circuits and Systems, pp. 586-589, August 2008, doi: 10.1109/ICECS.2008.4674921

[44] Xilinx. ZC702 Evaluation Board for the Zynq-7000 XC7Z020 SoC. [Online] Available at: https://www.xilinx.com/support/documentation/boards_and_kits/zc702_zvik/ug850-zc702-eval-bd.pdf

Joaquín Gracia-Morán received the B.Sc., M.Sc., and Ph.D. degrees in computer engineering from the Universitat Politècnica de València (UPV), Spain, in 1995, 1997, and 2004, respectively, where he is currently an Associate Professor with the Department of Computer Engineering (DISCA). His research interests include design and implementation of digital systems, design and validation of fault-tolerant systems and VHDLbased fault injection. He is a member with the Fault-Tolerant Systems (STF) research line within the Institute ITACA from the UPV.

Luis-J. Saiz-Adalid received the M.Sc. and Ph.D. degrees in computer engineering from the Universitat Politècnica de València (UPV), Spain, in 1995 and 2015, respectively.

After 15 years in the industry (IBM, 1995-Celestica, 1998). He is currently an Associate Professor with the DISCA, UPV. His research interests

include design and implementation of digital systems, design and validation of fault-tolerant systems and design of error control codes. He is a member with the Fault-Tolerant Systems (STF) research line within the Institute ITACA from the UPV.

J.-Carlos Baraza-Calvo received the B.Sc. and Ph.D. degrees in computer engineering from the Universitat Politècnica de València, (UPV), Spain, in 1993 and 2003, respectively. He is currently an Associate Professor with the DISCA. His research interests include design and implementation of digital systems, design and validation of fault-tolerant systems and VHDL-based fault injection. He is a member with the Fault-Tolerant Systems (STF) research line within the Institute ITACA from the UPV.

Daniel Gil-Tomás received the B.Sc. degree in electrical and electronic physics from the Universitat de València, Spain, in 1985, and the Ph.D. degree in computer engineering from the Universitat Politècnica de València (UPV), Spain, in 1999, where he is currently an Associate Professor with the DISCA. His research interests include design and validation of fault-tolerant systems, reliability physics and reliability of emerging nanotechnologies. He is a member with the Fault-Tolerant Systems (STF) research line within the Institute ITACA from the UPV.

Pedro-J. Gil-Vicente (M'93) received the B.Sc. degree in electronic engineering from the Universitat Politècnica de Catalunya (UPC), Tarragona, Spain, in 1979, and the M.Sc. degree in industrial engineering and the Ph.D. degree in computer engineering from the Universitat Politècnica de València (UPV), Spain, in 1985 and 1992, respectively, where he is currently a Professor with the DISCA. He has been the Head of the DISCA, and he is also the Head of the Fault-Tolerant Systems (STF) research line, within the Institute ITACA. His research interests include the design and validation of realtime fault-tolerant distributed systems, the dependability validation using fault injection, the design and verification of embedded systems, and the dependability and security benchmarking. He has authored more than 150 research articles on these subjects.

Prof. Gil-Vicente has also served as a Program Committee member in the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), the European Dependable Computing Conference (EDCC) and the Latin American Symposium on Dependable Computing (LADC).