Automated synthesis of EDACs for FLASH Memories with User-Selectable Correction Capability

Maurizio CARAMIA$^{(+)}$, Michele FABIANO$^{(*)}$, Andrea MIELE$^{(*)}$, Roberto PIAZZA$^{(*)}$, Paolo PRINETTO$^{(*)}$

(+)
Thales Alenia Space Italia
Command Control & Data Handling
-
{Maurizio.Caramia}thalesaleniaspace.com

(*)
Politecnico di Torino
Dipartimento di Automatica ed Informatica (DAUIN)
-
{Michele.Fabiano, Andrea.Miele, Paolo.Prinetto}@polito.it
{Roberto.Piazza}@studenti.polito.it

***Abstract*—Tackling the design of a mission-critical system is a rather complex task: different and quite often contrasting dimensions need to be explored and the related trade-offs need to be evaluated. Designing a mass-memory device is one of the typical issues of mission-critical applications: the whole system is expected to accomplish a high level of dependability which highly relies on the dependability provided by the mass-memory device itself. NAND flash-memories could be used for this goal: in fact on the one hand they are nonvolatile, shock-resistant and powereconomic but on the other hand they have several drawbacks (e.g., higher cost and number of erasure cycles bounded). Error Detection And Correction (EDAC) techniques could be exploited to improve dependability of flash-memory devices: in particular binary Bose and Ray-Chaudhuri (BCH) codes are a well known correcting code technique for NAND flash-memories. In spite of the importance of error correction capability several other equally critical dimensions need to be explored during the design of binary BCH codes for a flash-memory based mass-memory device. No systematic approach has so far been proposed to consider them all as a whole: as a consequence a novel design environment with a user-selectable error correction capability is aimed at supporting the design of binary BCH codes for a flash-memory based mass-memory device.**

I. Introduction

The expected level of dependability of mission-critical systems is always increasing: stricter requirements and more severe constraints are contributing to expand and complicate the whole design dimensions. In fact designing a mission-critical system is a rather complex task: different and quite often contrasting dimensions need to be explored and the related trade-offs need to be evaluated.

Designing a mass-memory device is one of the typical issues of mission-critical applications (e.g., for space applications [7]): the whole system is expected to accomplish a high level of dependability which highly relies on the dependability provided by the mass-memory device itself.

NAND flash-memories could be used for this goal: in fact NAND flash-memory based systems are gaining acceptance and usage not only in the consumer market but in mission-critical applications, as well, where they mainly play the role of high-capacity storage devices.

On the one hand flash-memories guarantee both the non-volatility in case of power loss and a highest storage density as well as they are shock-resistant and power-economic, but on the other hand they have several drawbacks [11] (e.g., higher cost and number of erasure cycles bounded). As a result, designing flash-based systems for mission-critical application requires both exploring a huge number of design dimensions and evaluating a huge amount of tradeoffs among all such dimensions [6].

Error Detection And Correction (EDAC) is respectively the ability to detect the presence of errors and to correct them: EDAC techniques could be exploited to improve the dependability of flash-memory devices. Designers should evaluate the most proper choice for their design, addressing many issues: the most significant ones include evaluating the type of code to adopt, choosing the number of bits needed for that code (i.e., for accomplishing the requested level of dependability) and addressing where that code has to be stored.

Binary Bose and Ray-Chaudhuri (BCH) codes [1], [10], [15] are a well known correcting code technique for NAND flash-memories. In general error correction capability is defined as the number of errors that a particular error correcting code (ECC) is able to correct: this is usually fixed by the requirements of the mission and is intuitively strongly linked to the computational power required to accomplish it (i.e., a higher error correction capability turns out in a more complex design). A user-selectable error correction capability would be the right tradeoff between the complexity and the requested level of dependability of the design: in fact the EDAC design would dynamically adapt to the current state of the mission-critical mass-memory device.

In spite of the importance of error correction capability several other equally critical dimensions need to be explored during the design of binary BCH codes for a flash-memory based mass-memory device. No systematic approach has so far been proposed to consider them all as a whole.

This paper presents the architecture of a novel design environment aimed at supporting the design of binary BCH codes for a flash-memory based mass-memory device with a user-selectable error correction capability. The project is mainly pushed by the unavailability, at our best knowledge, of a commercial tool capable of supporting a systematic analysis and exploration of the different possible alternatives with the chance of exploiting the advantages of a user-selectable error correction capability. Moreover this project is intended to be integrated with our FLARE design environment [5].

The rest of the paper is organized as follows: Section II addresses the main ideas related to the design of ECC, Section III explains in detail which are the main motivations for this work, focusing the attention also on the tradeoff between efficiency and complexity, Section IV proposes a possible architecture for a design environment aimed at supporting the design of binary BCH codes for a flash-memory based mass-memory device with a user-selectable error correction capability, while Section VI deals with a particular case study and presents the related experimental results.

II. Error Correcting Code Design

EDAC techniques are able to improve the dependability of flash-based device. Flash-memories presents several critical issues [4], [11] and the mission environment could strongly affects the design of a flash-based mass-memory device (e.g., space environment presents various issues especially because of radiations [6]): designing an ECC implies also to take care about all these issues.

Addressing the main basic code principles is absolutely needed: in order to accomplish this task, we will refer to linear ECC and, in particular, to binary BCH codes [13], [15]. This choice let us to be general and practical at the same time: in fact BCH codes presents both a design flow basically identical to the more general linear codes design flow and are practically adopted for on-chip error correcting [13]. Figure 3 resumes the BCH codes design flow.

Other linear ECC are represented by Hamming codes and Reed Solomon: several ECC algorithms for error checking and correction of NAND flash were proposed, based on Hamming codes or on Reed-Solomon codes [8], [16], [19] but are not addressed in the sequel of the paper.

A. Principles of error correcting codes

In spite of the several possible implementations of a particular ECC (e.g., BCH codes), the basic principle of all possible ECCs is fairly simple. Assuming data as strings of k bits in the binary domain, ECC algorithms first:

- consider a data string j_1 of k bits;
- convert (i.e., *encode*) these k bits in a new string (i.e., *codeword*) of n bits, with $n > k$;

Thus *encoding* operation is a one to one transformation that maps each k bits data to a n bits codeword.

Finally the codeword (i.e., n bits) is stored in the memory: all the possible error sources of the physical system can now affect the stored codeword.

When a codeword is read out from the memory, ECC algorithms:

- consider data of n bits;
- convert (i.e., *decode*) these n bits in a "new" string j_2 of k bits;

Thus *decoding* operation is intuitively dual to *encoding* one: it performs a transformation that maps each n bits data to a k bits codeword. At this point two data strings $j_1$and j_2 of k bits each show up: a metric to determine the possible differences (i.e., errors) between them is absolutely needed.

Figure 2 simply shows the codewords space. A *distance* between two binary strings of the same length can be defined as the number of different bits between them: the distance (i.e., the number of different bits) between two strings is defined as the Hamming distance d. Moreover the minimum distance d_{min} of a code is the Hamming distance of the pair of codewords with the smallest Hamming distance [3]. Different ECC algorithms exist for different Hamming distance d: intuitively they are able to guarantee a sort of a margin around each codeword in terms of d. This margin is referred as the *error correction capability* or t.

Therefore $j_1$and $j_2$would have a specific Hamming distance d and these errors could be corrected according to the t of the ECC algorithm used for the *encoding* and the *decoding* phases: errors can be corrected if $d_{min} \geq 2t + 1$ [10], otherwise in most cases at least the presence of uncorrectable errors can be detected.

Therefore knowing a priori the maximum number of errors in a codeword the errors can be always corrected.

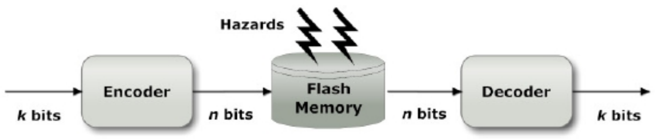

Figure 1. General Encoding/Decoding structure for Flash-memories

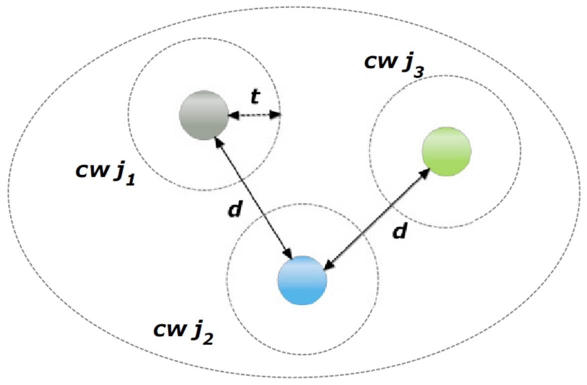

Figure 2. Codewords Space

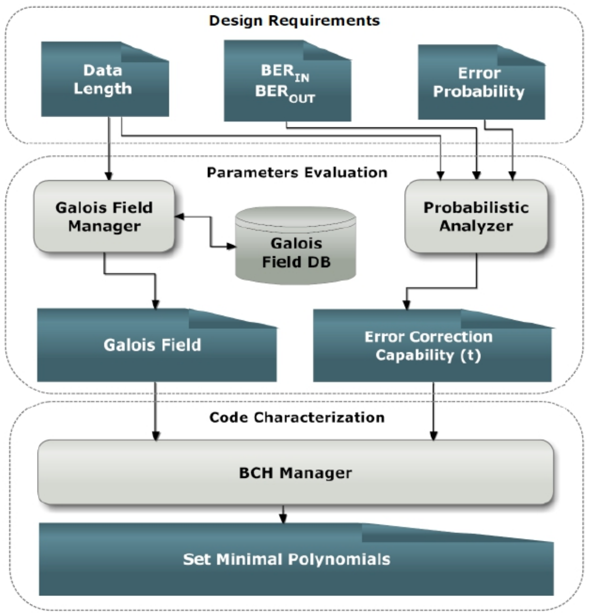

Figure 3. BCH Code Design Flow

To summarize, $r = n - k$ check bits are added to the k bits information data during the *encoding* phase (i.e., when data is *written*), while the r check bits and the k data bits are combined together in order to reconstruct the most probable information data during the *decoding* phase (i.e., when data is *read*): Figure 1 clearly depicts the two phases.

B. BCH Codes Design Flow

Figure 3 resumes the BCH codes design flow. Three main functional steps compose the design flow: firstly there is an initial phase of selection of the *Design Requirements*, which are the inputs for the following *Parameters Evaluation* phase; this second phase, in turn, evaluates and provides the inputs for the final *Code Characterization* phase, after which the BCH code is completely defined. Each step is going to be analyzed in the sequel of this section.

1) Design Requirements: The first step of each BCH code design flow is to define the mission-critical *requirements*: each particular mission would present its own specific requirements in terms of *data length of information* and of *probability of errors*. In other words each ECC algorithm works on data of fixed length (i.e., *Data Length*) and its correction capability would be determined according to probabilistic studies.

Let k be the data length, r the check bits and $n = k + r$ the codeword length; let assume also $r \ll k$ for simplicity (i.e., $n \approx k$). Finally let assume that each bit (i.e., the *i-th* physical cell) of a stored codeword has a p probability of error. Assuming a generic error correction capability t, a system failure turns out when more than t errors occur: in fact our system is able to correct up to t errors.

The probability of failure of at least w cells (i.e., at least w errors) could be expressed in terms of the probability of error p_i of the i-th cell:

$$P_{w/n} = \sum_{i=w}^{n} \binom{n}{i} p^i (1-p)^{n-i} \tag{1}$$

where Equation 1 sums all the probabilities to have more than w errors up to n errors, considering all the *i-th* cell (i.e., error) with $i = w, w+1, ..., n$. The probability to have at least one error is expressed by $w = 1$: this probability simply is the complement of the probability of having no errors in all the n cells of the page. In fact it could be expressed by:

$$P_{1/n} = 1 - (1-p)^n = P_e \tag{2}$$

In order to characterize our code, the so called *Bit Error Rate (BER)* has to be evaluated: in particular this parameter could be split in an *Input BER* or $BER_{in}$and an *Output BER* or BER_{out}, which are respectively the probability of error of the i-th cell of the flash and the desired probability of failure of the system. BER_{in} can be simply expressed as:

$$BER_{in} = p \tag{3}$$

BER_{out} is the probability of having more than t errors and can be compute through Equation 1 as:

$$BER_{out} = P_{t+1/n} = \sum_{i=t+1}^{n} \binom{n}{i} p^i (1-p)^{n-i} \approx$$

$$\approx \binom{n}{t+1} p^{t+1} (1-p)^{n-t-1} \tag{4}$$

where let assume $n \cdot p \ll 1$ in the last approximation of Equation 4. Equation 4 is the direct link between BER_{in} and BER_{out}: in fact Equation 4 let designers correctly choose t according both to the required probability of failure (i.e., BER_{out}) and to a rough estimation of p (i.e., BER_{in}). [10]

E.g., let assume $k = 2^{14} = 16384 bits = 2 Kbytes$: Figure 4 shows the resulting $BER_{out} = P_{t+1/n}$ of Equation 4 in function of $BER_{in} = p$ (i.e., *cell error probability*) , with for $t = \{0, 1, 5, 10, 15\}$.

In spite of the small difference among the different values of the *correction capability*, Figure 4 shows the evident overall

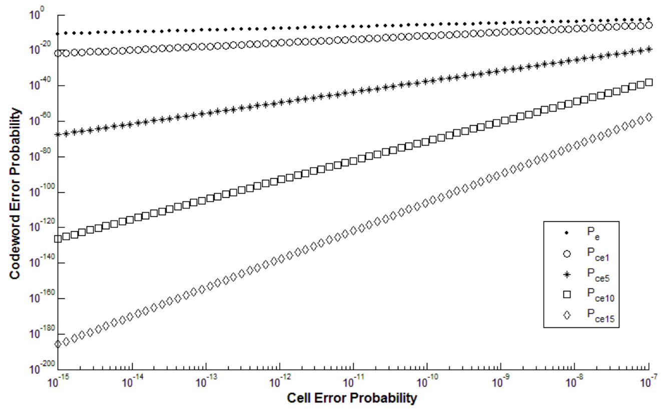

Figure 4. BER_{out} Vs BER_{in}

Table I
SHORT BCH VS LONG BCH CODES

	Short BCH	**Long BCH**
Hardware Implementation	<u>Simple</u>	Complex
Resources Overhead	<u>Low</u>	High
Error correction capability	Limited Range	<u>Wide Range</u>
Code Efficiency	Low	<u>High</u>

Table II
BCH CODE PROPERTIES

Specified by	zeroes $\alpha, \alpha^2, \alpha^3, ..., \alpha^{2t}$of all the codewords $w(x)$
Codewords Length	$n = 2^m - 1$
Information Symbols	$k = n - degree$ of the generator polynomial $g(x)$
Minimum Distance	$d \geq 2t - 1$
Error Control Capability	Corrects t errors

effect on the desired probability of failure (i.e., BER_{out}). Figure 4 will be exploited in Section II-B2 for choosing the most suitable *t* according to the estimated BER_{in} and the desired BER_{out} [10].

2) Parameters Evaluation: Let assume that the *Data Length k*, the BER_{in} and the desired BER_{out} have been defined according to the mission-critical dependability requirements. The second step takes these inputs and performs two main elaborations: the choices of the *Error Correction Capability t* and of the *Galois Field*. While *Error Correction Capability* has been addressed in Section II-A as the number of errors the ECC algorithm is able to correct, Galois Fields have not been discussed yet.

BCH codes are based on the abstract algebra and, in particular, on Galois Fields [1]. Basically a Galois Field for a specified *m*:

- contains 2^m elements, defined as $p_m(x = \alpha) = 0 \iff \alpha^m = b_{m-1}a^{m-1} + b_{m-2}a^{m-2} + ... + b_0$;
- all elements can be expressed as α^i with $i\epsilon\,(0, ..., 2^m - 2)$;
- always $\alpha^{2^m-1} = 1 = \alpha^0$;
- is closed respect addition and multiplication;

Each BCH code related to the specified *Data Length* needs a particular Galois Field to be generated.[15] shows a relation between *codeword length* (i.e., *data length k* and *check bits r*) and the *m* of the Galois Field:

$$2^q + r \leq 2^m - 1 \tag{5}$$

E.g., let assume $k = 2^q = 2^{14} = 16384bits = 2KBytes$: this implies a Galois Field with $2^m = 2^{15} = 32767$ elements, in order to satisfy Equation 6.

3) Code Characterization: Let assume, at this point, that the *Error Correction Capability* has been evaluated and that the *Galois Field* related to the data length has been generated. The third final step simply takes these inputs and generates the so called *Minimal Polynomials* $\psi_1(x), \psi_2(x), ..., \psi_{2t}(x)$ [1], [15] : they fully characterize the BCH error correcting code with respect to the defined inputs.

The set of *Minimal Polynomials*, in turns, defines the so called *Polynomial Generator g(x)* of the BCH code [1], which can be expressed as:

$$g\left(x\right) = LCM\left(\psi_1\left(x\right), \psi_2\left(x\right) ..., \psi_{2t}\left(x\right)\right) \tag{6}$$

where LCM is the *Least Common Multiple* operator among the *2t* minimal polynomials defined above.

Several applications of BCH codes have been proposed [9], [10], [13], [20]: Table II summarize the main BCH code properties.

4) Code length Vs Code efficiency: All the design choices could not be taken only with the ideas explained in Sections II-B1,II-B2 and II-B3: addressing the practical implementation of BCH codes is absolutely needed. Table I resumes the main advantages and drawbacks of implementing a short or a long BCH code: intuitively a shorter code is easier to implement, less resource-consuming, but with a poor efficiency, while a longer code is highly efficient, but more complex to implement and more resources-consuming. Table I shows the main tradeoffs between short and long BCH codes [15].

III. MOTIVATIONS

In this section the motivations of this work are presented. The target problem is the development of a real error correction system suitable for a NAND flash-memory based mass-memory device. In particular, the goal is to devise a powerful design environment for exploiting binary BCH codes: this environment is strongly intended to be *user-Selectable*, *Automatic* and *Parametric*.

Firstly it is intended to be fully user-selectable in terms of error correction capability: this need is stricly related to the disturbances [12], [17], [18] and to the reliability characterization of flash-memories [4], [11] as well as to the optimization of the overall performances of the whole system. On the one hand Flash-based devices are subject to *aging* (i.e., the write/read operations cycling): due to this phenomenon, after a certain number of cycles, the memory block becomes rapidly fully unreliable and should be discarded. On the other hand an important aspect of such a selectable rate correction system is the possibility to trade-off run time performance (i.e., in terms of *latency*) versus reliability (in terms of *correction capability*). Moreover the adaptivity of our solution can also be exploited to allow the correction of both transient and permanent faults [17].

Secondly our design environment is intended to be automatic and parametric at the same time: in particular an automatic hardware generation tool has been developed to tackle the high complexity of the BCH encoder and decoder. In addition this tool is fully parametric: this feature allows the exploration of the design space and the related tradeoffs.

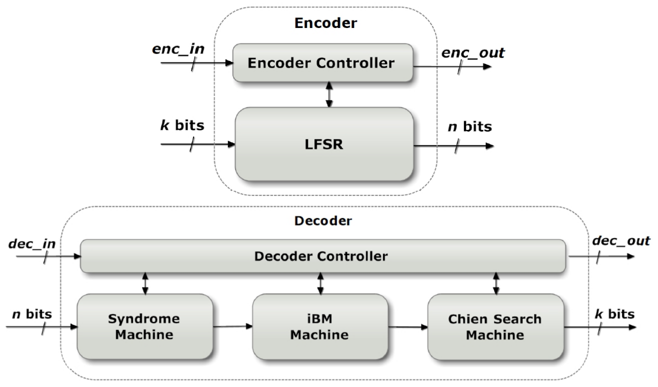

Figure 5. High Level Composing Blocks of Encoder and Decoder

A powerful BCH-based pipelined solution has been proposed in [9] and also [14] provides useful support for high-level design, but they both provides no adaptability features. The presented tool is intended not only to be fully selectable in terms of correction capability but is also able to tune several parameters related to the overall desired design: in particular parameters could be considered as the ones discussed in Subsection II-B1 and previously showed in Figure 3. In addition parameters that are strictly related to the hardware parallelism need to be addressed: in spite of the presence of parameters related both to the hardware and the software, most of them concern the hardware blocks of the whole system.

IV. The ADAGE Design Environment

In this section the proposed ADaptive ECC Automatic GEnerator (ADAGE) design environment with a user-selectable error correction capability is presented: ADAGE is aimed at supporting the design of binary BCH codes for a flash-memory based mass-memory device.

A. Overall Hardware Architecture

The general Figure 1 can be expanded in order to get a more accurate overall hardware architecture.

Figure 5 shows the composing blocks of the *Encoder* in Figure 1: it is mainly composed by a *parallel controllable LSFR* [9], [15] and an *Encoder Controller*. The first is simply a calculus machine receiving an input of *k bits* input and producing an output of *n bits*, while the second manages a set of control signals and controls the LFSR.

Figure 5 shows the composing blocks of the *Decoder* in Figure 1: it is composed by a *Syndrome Machine*, a *iBM Machine* [21], a *Chien Search Machine* [9], [15] and, finally, a *Decoder Controller*. The first three machines perform all

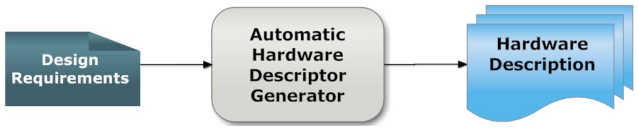

Figure 6. High Level View ADAGE tool

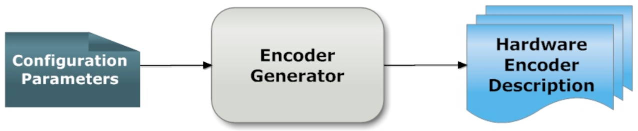

Figure 7. Encoder Generator High Level View

the needed computations for the decoding, while the last one schedules the operations and handles failure events.

B. Tool Architecture

Figure 6 shows the high-level view of ADAGE: ADAGE takes the *Design Requirements* addressed in Section II-B1 as inputs and is able to automatically and parametrically generate and connect all the blocks of the Adaptive BCH-based correction system.

The general Figure 1 can be exploited to understand that two main functional blocks are generated: the *Encoder* and the *Decoder*.

1) Encoder Generation: Figure 7 shows the high-level view of the *Encoder Generator*: it basically takes a set of *Configuration Parameters* as inputs, elaborates them and automatically provides a *HW Encoder Description*. The *Encoder Generator* could produce different outputs, according to a particular set of configuration parameters.

The set of parameters for the *Encoder Generator* is composed by the following elements: the *code length*, the *input parallelism*, t_{min}, t_{max}.

The *code length* affects the whole managing of the input data stream in the encoder. The *input parallelism* obviously matches with the parallelism of the hardware (i.e., a controllable LFSR) needed in the encoder for producing the codeword.

The t_{max} parameter is combined with the selected Galois Field (i.e., the functional “BCH Manager” of Figure 3) and affects the complexity (i.e., the number of internal flip-flops) of the resulting generated encoder: in particular, assuming a Galois Field of 2^m elements, the complexity grows with $m \cdot t_{max}$.

2) Decoder Generation: The high-level view of the *Decoder Generator* is functionally identical to the one in Figure 7 of the encoder: it basically takes a set of *Configuration Parameters* as inputs, elaborates them and automatically provides a *HW Decoder Description*. The *Decoder Generator* could produce different outputs, according to a particular set of configuration parameters.

The set of parameters for the *Decoder Generator* is composed by the following elements: the *code length*, the *input parallelism*, t_{min}, t_{max} and, in addition, the *Chien Search Parallelism* and the *iBM Parallelism*. The *code length*, the *input parallelism*, t_{min}, t_{max} are identical to the ones described in Subsection IV-B1.

The *iBM Parallelism* is an important parameter: it basically determines how the iBM Machine of Figure 5 is practically implemented. Designers are allowed to specify a *Low* or

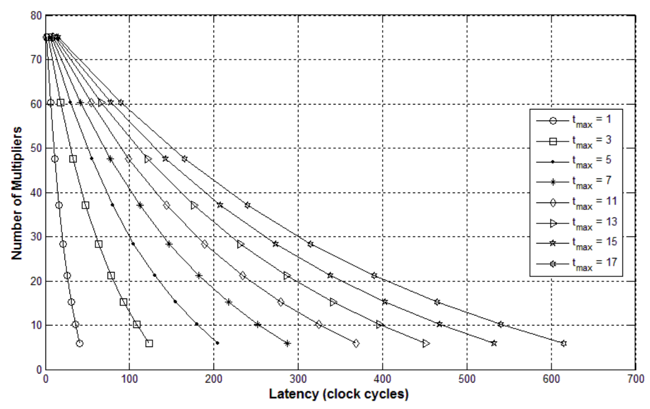

Figure 8. Latency Vs Complexity in iBM Machine

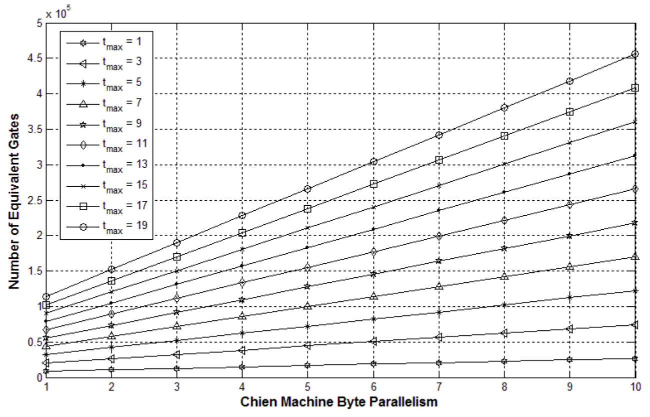

Figure 10. Chien Machine Parallelism Vs Equivalent Gates

High level of parallelism: it respectively implies the use of a *Sequential Multiplications iBM Machine* and of a *Full Parallel iBM Machine*. This tradeoff is addressed in Section IV-C1.

The spite the importance of iBM parallelism, the *Chien Search Parallelism* is a more critical parameter: in fact the Chien Search of Figure 5 is the most time consuming operation and presents a crucial tradeoff between latency and area. This tradeoff will be addressed in Section IV-C2.

C. Design Space Exploration

In the sequel of this section, the ADAGE tool will be exploited to explore the main dimensions of the design space addressed in Section IV-A.

1) iBM Parallelism: Sequential Multiplications iBM Machine and *Full Parallel iBM Machine* are two different implementations of the iBM Machine. They differ in the way of performing multiplications: this implies the *Sequential iBM* to be slower but less area consuming, whereas the *Parallel iBM* provides high-performances but with a wider area. Figure 8 shows the relation between iBM latency and complexity for several values of t_{max}: the iBM latency is defined as the time between the arrival of the the information from the Syndrome Machine and the dispatch of the evaluated data to the Chien Machine.

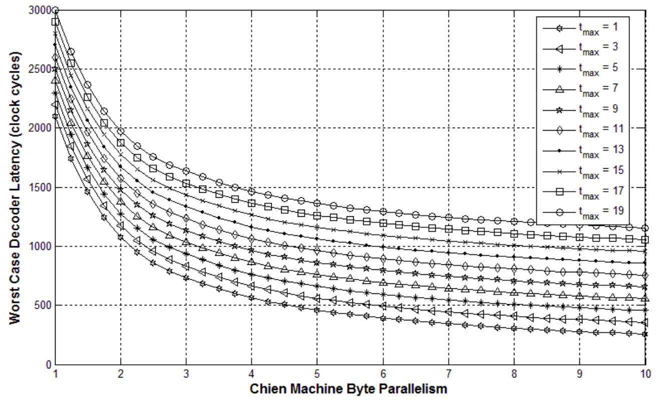

Figure 9. Chien Machine Parallelism Vs Decoder Latency

2) Chien Machine Parallelism: Firstly let assume the use of the so called *Double Chien Machine* [15]: in fact it is the most suitable basic tradeoff between performance and resources. Secondly designers are able to set its *parallelism* (or *p*): the Double Chien Machine will simply consider *p* bytes at each clock cycle. At this point designers have to evaluate the related tradeoffs. E.g., considering several values of t_{max}, Figure 9 shows the relation between *p* and the resulting decoder *latency*, while Figure 10 shows the relation between *p* and the resulting *area*.

D. Example: a Chien Machine Generation

The spite it is the most critical and complex part of the whole ECC system, ADAGE is able to easily generate a complete Chien Machine. Let assume the *Single Chien Machine* case with $t_{max} = 15$ and $p = 8$: ADAGE generates a Chien Machine made by $(t_{max} - 1) = 14$ 15-bits registers, $(t_{max} - 1) = 14$ 15-bits multiplexers, $(t_{max} - 1) = 14$ full multiplexers, $(t_{max} - 1) \cdot p = 14 \cdot 8 = 112$ constant multipliers, $(t_{max} - 1) \cdot p = 112$ XOR gates. In the *Double Chien Machine* case, each machine is generated independently with its own t_{max_1}, t_{max_2}, p_1 and p_2: ADAGE is able to automatically connect them and to provide the proper control signals to handle them.

V. Validation and Verification

A proper environment has been designed to Validate and Verify our ADAGE tool. Figure 11 shows the architecture of

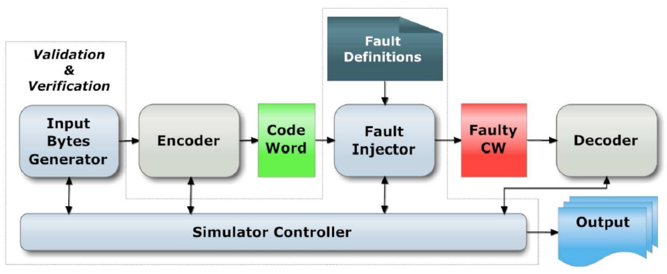

Figure 11. V&V Architecture

Table III
CASE STUDY ARCHITECTURES

	iBM Machine	Chien Machine
Architecture 1	*Parallel*	$p = 8$
Architecture 2	*Sequential*	$p = 8$
Architecture 3	*Sequential*	$p = 48$
Architecture 4	*Sequential*	*Double Chien*

the V&V environment: it is a high-level simulator providing the input data generation, the flash-memory emulation, the fault injection and the storage in a file of the collected decoder outputs.

V&V was done via a significant fault-injection campaign. In particular the simulation cycle is defined as follows:

1) The set of faults to be injected in the codeword are defined
2) For a correction capability *t* from 1 to 15
    a) The input data in *Encoded* (i.e., codeword);
    b) The *faulty codeword* is generated with the help of the *faults definitions;*
    c) The *faulty codeword* is *Decoded* and the output of the decoder are collected and written in the *Output* file;

The FARM model was used [2]. Bit flippings were randomly injected and their number dynamically varied to span the complete set of possible concurrent errors. The Readout was performed resorting to external files.

VI. CASE STUDY

Let assume the *data length* $k = 2^{14} = 16384 = 4KBytes$, $t_{max} = 15$, $t_{min} = 1$ and a *data width* of 1Bytes. The architectures listed in Table III have been addressed.

A. V&V Experimental Results

The simulations were aimed at both verifying the correctness of the design and at measuring the latencies in different situations. The correctness has been verified injecting faults as described in Section V and then launching simulation cycles: in particular for each simulation cycle we stored in a file the results corresponding to the tested correction capabilities.

An example of an output file, in which six errors in different positions were injected, is reported below:

```
END ECC t =  1: Timer = 2097; Failure = 1
END ECC t =  2: Timer = 2140; Failure = 1
 -> ECC t =  3 Found Error: Index = 2051 Mask = 00100000
END ECC t =  3: Timer = 2185; Failure = 1
 -> ECC t =  4 Found Error: Index =  294 Mask = 01000000
(...)
 -> ECC t = 14 Found Error: Index = 2049 Mask = 00101000
END ECC t = 14: Timer = 2749; Failure = 0
 -> ECC t = 15 Found Error: Index = 2048 Mask = 01000100
 -> ECC t = 15 Found Error: Index = 2049 Mask = 00101000
END ECC t = 15: Timer = 2810; Failure = 0
```

The latency is defined as the time between the moment in which memory outputs the last byte and the one in which the decoding ends (i.e., the time for decoding the data). The so

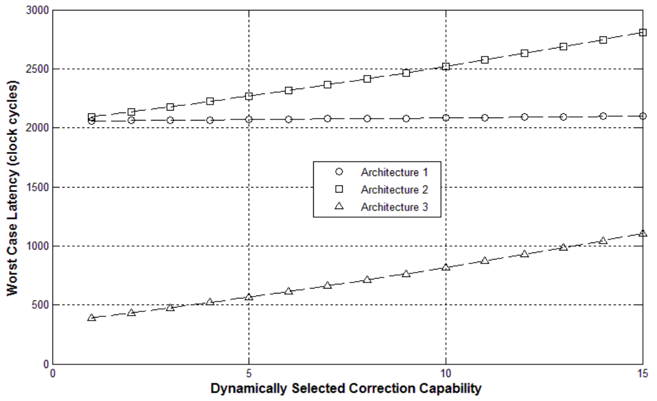

Figure 12. Single Chien Decoder Latencies

called *Worst Case Latency* occurs when $tmax$ is the maximum possible and depends on the position of the faulty bits: since faults were randomly injected, the response (i.e., the latency) is expected to be high or low according to the position of the injected bit-error.

Let assume to inject a single error: the bit error in the first and last bit position tested by the Chien machine respectively provide the *best* and the *worst* latency. The worst case latencies of the Architectures 1, 2 and 3 are plotted in Figure 12 in function of the selected correction capability *t*.

Architecture 4 needed a different approach: the adaptive error correction system will present different latencies in function of the selected Chien machine (i.e., the first or the second one). Four errors in the first and last four bits were injected in order to respectively test the *best* and the *worst* latency for this Chien machine. The worst case latencies of the Architectures 4 are plotted in Figure 13 in function of the selected correction capability *t*: the resulting latencies are the combined ones of the Architecture 2 and 3.

B. Synthesis Experimental Results

After verifying the correctness of the environment, the *Synopsys Design Vision Environment* has been exploited for

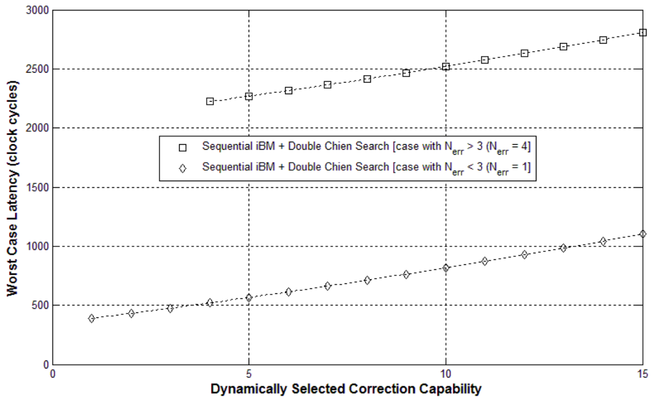

Figure 13. Double Chien Decoder Latencies

Table IV
COMPLEXITY AND PERFORMANCE ESTIMATION

	Equivalent Gate Count	**Clock Frequency (MHz)**
Architecture 1	170.963K	61.95
Architecture 2	96.698K	62.97
Architecture 3	221.360K	28.40
Architecture 4	135.194K	27.81

evaluating the complexity related to the considered Architecture of Table III. Table IV shows the complexity of each architecture in terms of equivalent number of gates and maximum achievable clock frequency.

Architecture 1 (i.e., a parallel iBM Machine and $p = 8$ Chien Machine) and Architecture 2 (i.e., same of Architecture 1 with a sequential iBM Machine) can be easily compared: the spite they implement the same Chien Machine, Architecture 1 provides a higher area (i.e., higher number of gates) but also a lower latency than Architecture 2 as Figure 13 clearly highlights.

Moreover Figure 13 also shows Architecture 3 to provide the lowest latency among the first three cases: however Table IV underlines that the complexity of the third case is the highest among all proposed architectures.

Finally Figure 13 and Table IV clearly show Architecture 4 to be the most suitable tradeoff among all the other architectures: in fact, in the average, it provides the same latency of the third case with an extremely reduced cost in terms of area.

VII. CONCLUSIONS

In this paper a parametric design environment for supporting the design of BCH codes for flash-memory based mass-memory device has been presented.

All the main problems of both the flash-memories and the correction codes design has been largely explored and taken in account. Our environment provides high flexibility to the designer and allows the tuning of the different parameters and trade-offs that arise designing a BCH codes based ECC system. The user-selection feature has been widely motivated and together with the characteristics explained above represents a significant element of novelty, not offered by previous works. The effectiveness of our solution has been proved by the experimental results which confirm that it represents a significant aid for solving the problems of the ECC hardware systems design.

In our future works the ADAGE environment will be exploited for an integration between EDAC and Testing strategies: in fact Testing strategies could be exploited in order to understand the actual state of the system and, in turns, the EDAC structure should be able to adapt its own error correction capability according to the results provided by the test environment. Integrating these two main aspects (i.e., EDAC and Testing) would result in providing both a adaptive-rate correction system and a fault-tolerant mass-memory device.

REFERENCES

[1] J. Adamek. *Foundations of Coding: Theory and Applications of Error-Correcting Codes, with an Introduction to Cryptography and Informat.* John Wiley & Sons, Inc., New York, NY, USA, 1991.

[2] A. Benso and P. Prinetto. *Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation*, volume 1-4020-7589-8. Kluver Academic, 2003.

[3] Richard E. Blahut. *Theory and Practice of Error Control Codes.* Addison-Wesley.

[4] J. Brewer and M. Gill. *Nonvolatile Memory Technologies with Emphasis on Flash: A Comprehensive Guide to Understanding and Using Flash Memory Devices*, volume 978-0-471-77002-2. Wiley-IEEE Press, February 2008.

[5] M. Caramia, S. Di Carlo, M. Fabiano, and P. Prinetto. Flare: A design environment for flash-based space applications. *Proceedings of High Level Design Validation and Test Workshop, 2009. HLDVT 2009. IEEE International*, pages 14 –19, nov. 2009.

[6] M. Caramia, S. Di Carlo, M. Fabiano, and P. Prinetto. Flash-memories in space applications: Trends and challenges. *Proceedings of East-West Design & Test Symposium (EWDTS)*, pages 18–21, September 2009.

[7] M. Cassel, D. Walter, H. Schmidt, F. Gliem, H. Michalik, M. Stähle, K. Vögele, and P. Casel Roos. Nand-flash-memory technology in mass memory systems for space applications. *Proceedings Data Systems In Aerospace (DASIA) 2008*, 2008. Palma de Mallorca, Spain.

[8] S. Chen. What types of ecc should be used on flash-memory? Technical report, Spansion, November 2007.

[9] Te-Hsuan Chen, Yu-Ying Hsiao, Yu-Tsao Hsing, and Cheng-Wen Wu. An adaptive-rate error correction scheme for nand flash memory. *VLSI Test Symposium, 2009. VTS '09. 27th IEEE*, pages 53 –58, may 2009.

[10] S. Gregori, A. Cabrini, O. Khouri, and G. Torelli. On-chip error correcting techniques for new-generation flash memories. *Proceedings of the IEEE*, 91(4):602 – 616, april 2003.

[11] D. Ielmini. Reliability issues and modeling of flash and post-flash memory (invited paper). *Microelectronic Engineering*, 86(7-9):1870 – 1875, 2009. INFOS 2009.

[12] Jae-Duk Lee, Sung-Hoi Hur, and Jung-Dal Choi. Effects of floating-gate interference on nand flash memory cell operation. *Electron Device Letters, IEEE*, 23(5):264 –266, may 2002.

[13] Wei Liu, Junrye Rho, and Wonyong Sung. Low-power high-throughput bch error correction vlsi design for multi-level cell nand flash memories. *Signal Processing Systems Design and Implementation, 2006. SIPS '06. IEEE Workshop on*, pages 303 –308, oct. 2006.

[14] The MathWorks. http://www.mathworks.com/products/slhdlcoder/.

[15] R. Micheloni, A. Marelli, and R. Ravasio. *Error Correction Codes for Non-Volatile Memories.* Springer Publishing Company, Incorporated, 2008.

[16] Micron. Hamming codes for nand flash-memory devices overview. Technical Report 29-08, May 2007.

[17] M.G. Mohammad and K.K. Saluja. Flash memory disturbances: modeling and test. *VLSI Test Symposium, 19th IEEE Proceedings on. VTS 2001*, pages 218 –224, 2001.

[18] Mincheol Park, Keonsoo Kim, Jong-Ho Park, and Jeong-Hyuck Choi. Direct field effect of neighboring cell transistor on cell-to-cell interference of nand flash cell arrays. *Electron Device Letters, IEEE*, 30(2):174 –177, feb. 2009.

[19] Ltd Samsung Electronics Co. Nand flash ecc algorithm (error checking & correction). Technical report, June 2004.

[20] F. Sun, S. Devarajan, K. Rose, and T. Zhang. Design of on-chip error correction systems for multilevel nor and nand flash memories. *Circuits, Devices Systems, IET*, 1(3):241 –249, june 2007.

[21] Xu Youzhi. Implementation of berlekamp-massey algorithm without inversion. *Communications, Speech and Vision, IEE Proceedings I*, 138(3):138 –140, june 1991.