architecture is reported in Fig. 5.1. It enables to dynamically change the generator polynomial of the LFSR. This is a key feature in the implementation of an adaptable BCH encoder.

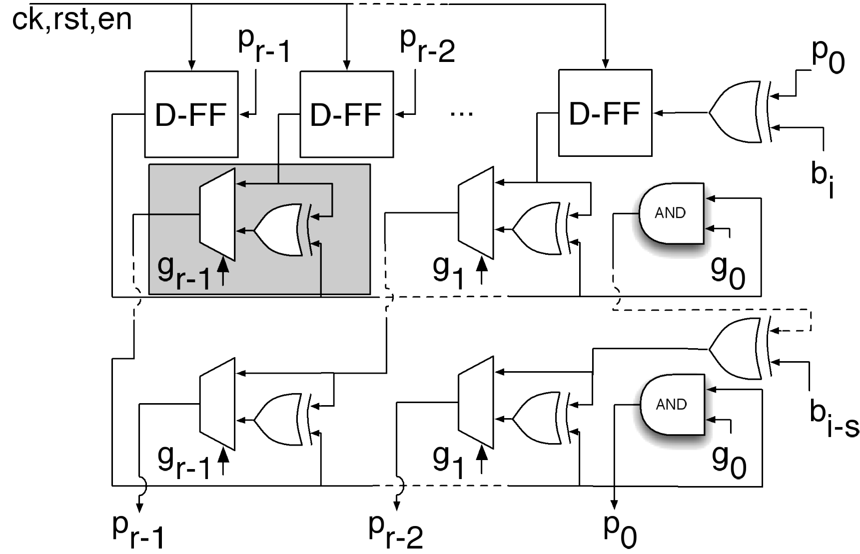


Figure 5.1: Architecture of a r-bit PPLFSR with s-bit parallelism [60].

The gray box of Fig. 5.1 highlights the basic adaptable block of this circuit. It exploits a multiplexer, controlled by one of the coefficients of the desired divisor polynomial, to dynamically insert an XOR gate at the output of one of the related D-type flip-flops composing the register. The s vertical stages of the circuit implement the parallelism of the PPLFSR computing the state at clock cycle $i+s$, based on the state at cycle i. However, this solution has high overhead. In fact, such PPLFSR is able to divide by all possible r-bit polynomials, while just well selected divisor polynomials are required.

Although Chen at al. deeply analyze the encoding process and the issues related to the storage of parity bits, the decoding process is scarcely analyzed, without providing details on how adaptability is achieved. Four different correction modes, namely $t = (9, 14, 19, 24)$ are considered in [37] for a BCH code defined on $GF(2^{13})$ with a block size of 512B (every 2 KB page of the flash is split in four blocks). The selection of the 4 modes is based on considerations about the number of parity bits to store. However, there is no provision to understand whether additional modes can be easily implemented. As an example, when selecting correction modes in which the size of the codeword is not a multiple of the parallelism of the decoder, alignment problems arise,

which are completely neglected in the paper.

5.2 Optimized Architectures of Programmable Parallel LFSRs

In this section, we will introduce an optimized block to perform an adaptable remainder computation. In fact, one of the most recurring operations in BCH encoding/decoding is the remainder computation between a polynomial representing a message to encode/decode and a generator/minimal polynomial of the code, that depends on the selected correction capability. The PPLFSR of Fig. 5.1 can perform this operation [37].

An r-bit PPLFSR can potentially divide by any r-bit polynomial by properly controlling its configuration signals ($g_0 \dots g_{r-1}$). However, in BCH encoding/decoding, even considering an adaptable codec, just well selected divisor polynomials are required (e.g., the generators polynomials $g_9(x)$, $g_{14}(x)$, $g_{19}(x)$, $g_{24}(x)$ of the four implemented correction modes of [37]). This computational block is therefore highly inefficient. Moreover, the set of divisor polynomials required in a BCH codec usually share common terms among each other. Such terms can be exploited to generate an optimized PPLFSR (OPPLFSR) architecture.

Let us consider, as an example, the design of a r=15-bit programmable LFSR able to divide by two polynomials $p_1(x) = x^{15} + x^{13} + x^{10} + x^5 + x^3 + x + 1$ and $p_2(x) = x^{13} + x^{12} + x^{10} + x^5 + x^4 + x^3 + x^2 + x + 1$ using a s=8-bit parallelism.

A traditional PPFLSR implementation would require $15 \times 8 = 120$ gray boxes (i.e., 120 XORs-MUXs). According to this implementation, this PPLFSR could divide by any $2^{15} = 32,768$ possible 15-bit polynomials, even if just 2 polynomials (i.e., the 0.006% of its full potential) are required.

An analysis of the target divisor polynomials can be exploited to optimize the PPLFSR architecture. Table 5.1 reports the binary representation of the two polynomials.

In it, three categories of polynomial terms can be identified:

1. Common terms (represented in bold), i.e., terms defined in all considered polynomials (x^{13}, x^{10}, x^5, x^3, x, and 1 in Table 5.1). For these terms, an XOR will be always required in the PPLFSR, thus saving the area dedicated to the MUX and the related control logic.

2. Missing terms (represented in underlined italic zeros), i.e., terms not defined in any of the considered polynomials, (x^{14}, x^{11}, x^9, x^8, x^7 and x^6 in Table 5.1). For these

Table 5.1: An example of the representation of $p_1(x)$ and $p_2(x)$

	x^{15}	x^{14}	x^{13}	x^{12}	x^{11}	x^{10}	x^{9}	x^{8}	x^{7}	x^{6}	x^{5}	x^{4}	x^{3}	x^{2}	x^{1}	1
$p_1(x)$	1	$\underline{0}$	**1**	0	$\underline{0}$	**1**	$\underline{0}$	$\underline{0}$	$\underline{0}$	$\underline{0}$	**1**	0	**1**	0	**1**	**1**
$p_2(x)$	0	$\underline{0}$	**1**	1	$\underline{0}$	**1**	$\underline{0}$	$\underline{0}$	$\underline{0}$	$\underline{0}$	**1**	1	**1**	1	**1**	**1**

terms both the XOR and the related MUX can be avoided.

3. Specific terms, i.e., terms that are specific of a subset of the considered polynomials (x^{15}, x^{12}, x^4, x^2 in Table 5.1). These terms are the only ones actually required.

We can therefore implement an optimized programmable LFSR (OPPLFSR) with three main building blocks:

1. each common present term (i.e., columns of all "1" of Table 5.1) needs an XOR, only;

2. each common absent term (i.e., columns of all "0" of Table 5.1) needs neither XOR nor MUX;

3. each specific term has a gray box, as Fig. 5.1;

Fig. 5.2 shows the resulting design for the portion x^{15}, x^{14} and x^{13}.

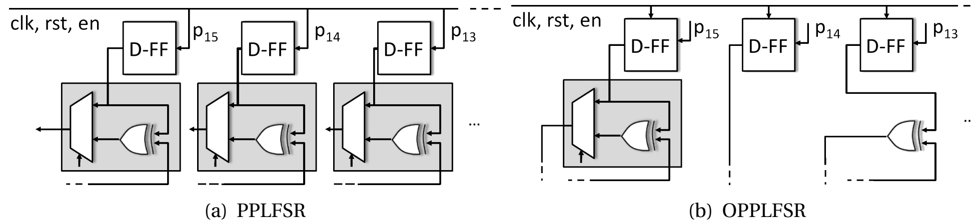


Figure 5.2: Example of the resulting PPLFSR (a) and OPPLFSR (b) with 8-bit parallelism for x^{15}, x^{14} and x^{13} of $p_1(x)$ and $p_2(x)$ [60]

This optimization also applies to polynomials with very different lengths. As an example, an OPPLFSR with single bit parallelism and able to divide by $p_1(x) = x^{225} + x + 1$ and $p_2(x) = x+1$, would only require a single adaptable block, compared to the 226 blocks required by a normal PPLFSR. Furthermore, the advantage of the OPPLFSR increases with the parallelism of the block. In fact, with the same 2 polynomials, a 8-bit OPPLFSR would require 8 adaptable blocks compared to $226 \times 8 = 1,808$ adaptable blocks of a traditional PPLFSR.

For sake of generality, Fig. 5.3 shows the high-level architecture of a generic OPPLFSR. Such a block is able to divide by a set $p_1(x), ..., p_M(x)$ of polynomials. We denote with q the number of required gray boxes.

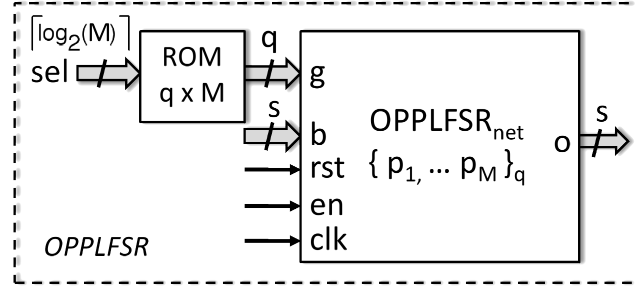


Figure 5.3: High-level architecture of the OPPLFSR [60]

The OPPLFSR interface includes: a s-bit input port (b) used to feed the data, a $\lceil \log_2(\mathrm{M}) \rceil$-bit input port (`sel`) used to select the polynomial of the division, and a s-bit port (o) providing the result of the division. Two blocks compose the OPPLFSR: *OPPLFSR$_{net}$* and *ROM*. The OPPLFSR$_{net}$ represents the complete network, partially shown in the example of Fig. 5.2. Given the output of the ROM, the q-bit signal g controls the MUXs of the q gray boxes (Fig. 5.2) according to the selected polynomial. The ROM is optimized accordingly with the design of the OPPLFSR, which leads to a reduced ROM and to a lower area overhead w.r.t. a full PPLFSR.

5.3 BCH Code Design Optimization

In this section, we address first the issue of choosing the most suitable set of polynomials for an optimized adaptable BCH code. Then, we propose a novel block, shared between the adaptable BCH encoder and the decoder, which reduces the area overhead of the resulting codec core.

5.3.1 The choice of the set of polynomials

The optimization offered by the OPPLFSR introduced in Section 5.2, may become ineffective if not properly exploited. It depends on the number and on the terms of the shared divisor polynomials implemented in the block. As an example, an excessive number of shared polynomials may make it difficult to find common terms, leading to an unwilled increase of the area overhead. Therefore, the choice of the polynomials to share is critical and must be properly tailored to the overall design.

Let us denote by Ω the set of t generators $g_i(x)$ and t minimal polynomials ψ_i which fully characterize an adaptable BCH code (see Section 5.1). Since for $GF(2^m)$ several

primitive polynomials $\psi_i(x)$ can be used to define the code, several set Ω_i can be constructed. Choosing the most suitable set Ω_i is critical to obtain an effective design of the OPPLFSR. On the one hand, it can be shown that the complexity of Ω_i increases with m [92, 118, 132]. On the other hand, the current trend is to adopt BCH codes with high values of m (e.g., $GF(2^{15})$) because current flash devices features a worse bit error rate [52]. Therefore, a simple visual inspection of each set Ω_i is not feasible to find the most suitable set of polynomials. An algorithmic approach is mandatory.

Each set Ω_i can be classified resorting to a *Maximum Correlation Index* (MCI). We define as $MCI(p_1, p_2, ..., p_N)$ the maximum number of common terms shared by a generic set of polynomials $p_1, p_2, ..., p_N$. As an example, the polynomials of Table 5.1 have $MCI(p_1, p_2) = 12$.

In the sequel, we introduce an algorithm to assess each set Ω_i according to its MCI. Given $i = \{1, ..., Y\}$, for each set Ω_i:

1. consider $\Omega_i = \{p_1, ..., p_N\}$ and $v_0 = p_1$;
2. determine the polynomial p_h such that the partition $S_{i,1} = (v_0, p_h)$ has the maximum $MCI(v_0, p_h)$, where $h = \{1, ..., N\}$ and $p_h \neq v_0$;
3. determine the polynomial p_k such that the partition $S_{i,1} = ((v_0, p_h), p_k)$ has the maximum $MCI(v_0, p_h, p_k)$, where $k = \{1, ..., N\}$ and $p_k \neq p_h \neq v_0$;
4. repeat step 3 until all polynomials have been considered in the partition $S_{i,1}$;
5. change the starting polynomial to the next one, e.g., $v_0 = p_2$, considering $S_{i,2}$ and repeat steps 2-4;
6. when $v_0 = p_N$, consider the next set Ω_{i+1}.

The algorithm ends when all sets Ω_i have been analyzed. For each Ω_i, the output is a set of partitions:

$$S_{i,j} = \{S_{i,1}, S_{i,2}, ..., S_{i,N}\} \tag{5.5}$$

Fig. 5.4 graphically shows the MCI of two partitions generated from two different starting points, for an hypothetical set Ω_i.

Fig. 5.4 shows that MCI always has a decreasing trend with the size of the partition S. This is straightforward since adding a polynomial may only decrease or keep constant

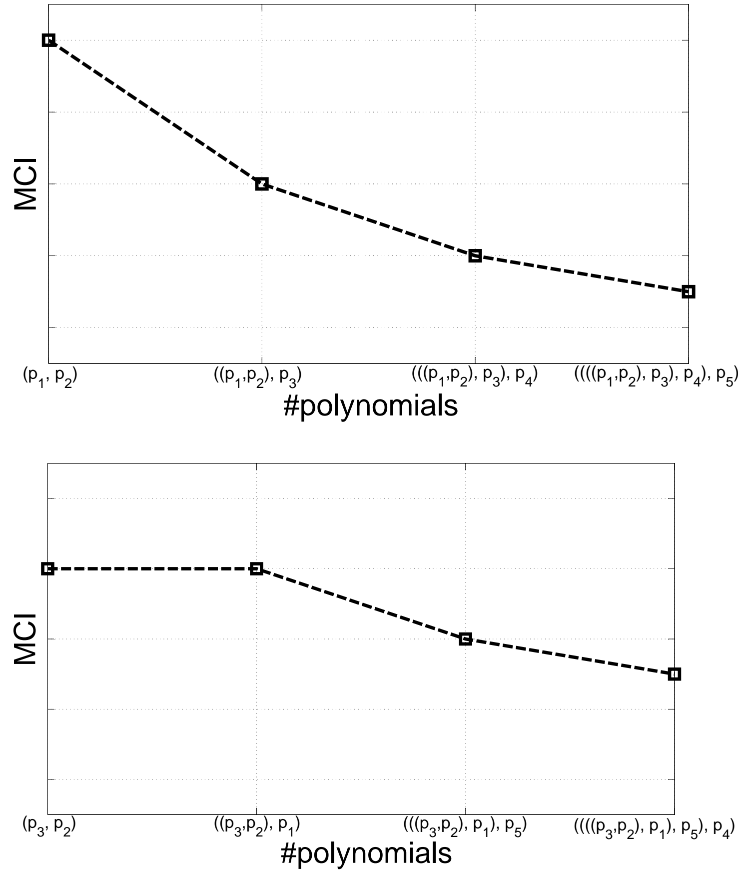


Figure 5.4: MCI examples of two hypothetical partitions $S_{i,1}$ and $S_{i,2}$

the current value of MCI. The curves, reported in 5.4, are critical in the choice of the most suitable set of polynomials for an optimized BCH code. For each partition $S_{i,j}$ with $j = \{1...N\}$, we can compute the average MCI (MCI_{avg}) as:

$$MCI_{avg}(S_{i,j}) = \frac{1}{N}\sum_{l=1}^{N-1} MCI_l \tag{5.6}$$

Eq. 5.6 applies to each set Ω_i where $i = \{1...Y\}$.

The best partition of the set Ω_i is then computed selecting the one with maximum MCI_{avg}:

$$S_{best_i} = \underset{j}{argmax}\left[MCI_{avg}\left(S_{i,j}\right)\right] \tag{5.7}$$

Finally, Eq. 5.8 compares the best partition of each set Ω_i to find the best set of polynomials:

$$S_{bestBCH} = \underset{i}{argmax}\left[S_{best_i}\right] \tag{5.8}$$

Eq. 5.8 defines the family of polynomials $S_{bestBCH}$, with the maximum average number of common terms.

Table 5.2: An example of Ω_i

	x^6	x^5	x^4	x^3	x^2	x^1	1
p_1	1	0	1	0	0	1	0
p_2	1	1	0	1	0	1	1
p_3	1	0	1	1	1	1	1
p_4	0	1	1	0	0	0	1
p_5	1	1	0	1	1	0	1
p_6	0	0	1	0	0	1	1

Let us provide an example to support the understanding of the algorithm. Suppose to consider a single set Ω_i composed of the polynomials of Table 5.2. The steps of the algorithm are:

1. Let us start with $v_0 = p_1$
2. We first evaluates $MCI\left(p_1, p_2\right) = 3$, $MCI\left(p_1, p_3\right) = 4$, $MCI\left(p_1, p_4\right) = 3$.
   Since $MCI\left(p_1, p_3\right) = 4$ is the maximum, the resulting partition is $S_{i,1} = \left\{p_1, p_3\right\}$
3. The next step considers $MCI(\left(p_1, p_3\right),\ p_2) = 3$ and $MCI(\left(p_1, p_3\right),\ p_4) = 3$. It is straightforward that the choice of either p_2 or p_4 does not affect the final value of the MCI_{avg}.

Given Ω_i with starting point p_1, it can be shown that the final partition is $S_{i,1} = \left\{\left(\left(p_1, p_3\right), p_4\right), p_2\right\}$ with a $MCI_{avg} = (4+3+3)/4 = 2.5$ from Eq. 5.6.

The complete algorithm iterates this computation for all possible starting points. Fig. 5.5 graphically shows the output of the MCI associated with each partition $S_{i,j}$ calculated for the following starting point $j = \{1, 2, 3, 4\}$.

According to Eq. 5.7, $S_{i,2}$ (the bold line) is the S_{best_i} of the example of Table 5.2, with a $MCI_{avg}\left(S_{i,j}\right) = 4$.

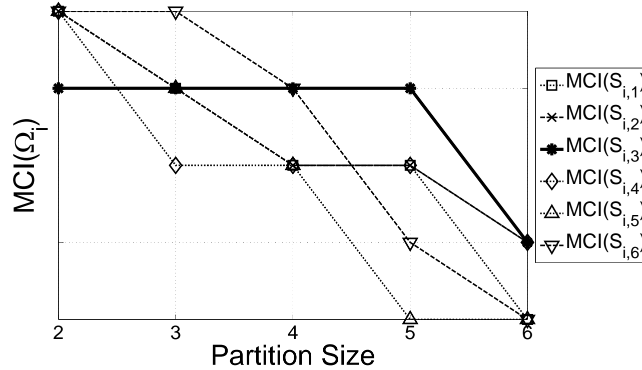


Figure 5.5: The MCI Trend of Table 5.2 [60]

5.3.2 Shared Optimized Programmable Parallel LFSRs

Let us assume to design an adaptable BCH code with correction capability from 1 up to t_M. Such a code needs to compute remainders of the division of:

- the message $m(x)$ by (potentially) all generator polynomials from g_1 up to g_{t_M}, for the encoding (5.2);
- the codeword $c(x)$ by (potentially) all minimal polynomials from $\psi_1(x)$ up to $\psi_{2t_M-1}(x)$, to compute the set of syndromes required during the decoding phase.

In a traditional implementation, these computations are performed by two separate set of LFSRs. In this chapter, we propose to devise a shared set of LFSRs able to: (i) perform all these computations, and (ii) reduce the overall cost in terms of resources overhead. Therefore, we can adopt the same shared set of LFSRs both in the encoding and decoding processes. This is possible since in a flash memory these operations are, in general, not required at the same time.

The OPPLFSR, introduced in Section 5.2, is the main building block of the set of shared LFSRs. Therefore, we will refer hereafter to such set of LFSRs as shared OPPLFSR (shOPPLFSR). Fig. 5.6 shows the high-level architecture of the shOPPLFSR. Its interface includes: a s-bit input port (`IN`) used to input the data to be divided, a $\lceil \log_2(\mathrm{N}) \rceil$-bit input port (`en`) used to enable each OPPLFSR, an input port (`sel`) used to select the proper polynomial by which each OPPLFSR has to divide, and a $\mathrm{N} \times$ s-bit port (p) providing the result of the division.

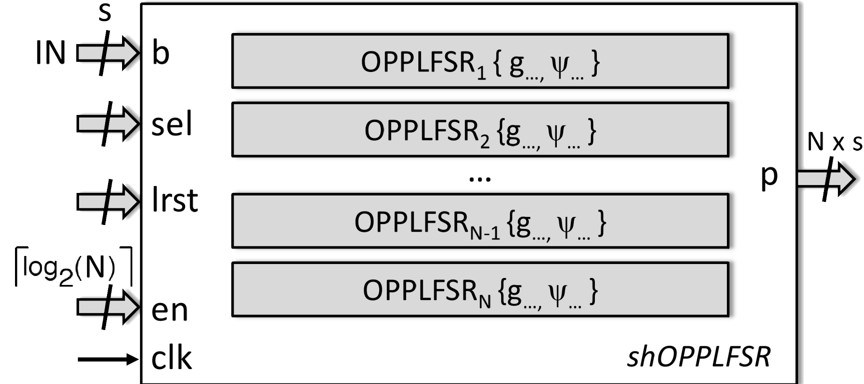


Figure 5.6: The shOPPLFSR architecture is composed by multiple OPPLFSRs

Given N OPPLFSRs and a maximum correction capability t_M, each OPPLFSR$_\mathrm{i}$ performs the division by a set of generator polynomials $g(x)$ and minimal polynomials $\psi(x)$. Such shOPPLFSR can be seen as an optimized programmable LFSR able to:

- divide by all generator polynomials from $g_1(x)$ to $g_{t_M}(x)$;
- divide by specific subsets of minimal polynomials from Eq. 5.1, as well.

An improper choice of the shared polynomials $g(x)$ and $\psi(x)$ can dramatically reduce the performance of the overall BCH codec. Also the partitioning strategy adopted is critical to maximize the optimization in terms of area, minimizing the impact on the latency of encoding/decoding operations.

The algorithm presented in Section 5.3.1 provides a valuable support for the exploration of this huge design space. In fact, the proposed method can be exploited to properly partition polynomials into the different OPPFLSRs of Fig. 5.6, in order to maximize the optimization of the resulting shOPPFLSR. Such optimization should not be obtained following blindly the outcomes of the algorithm, but always tailoring them to the specific design. Regarding this topic, Section 5.6 provides more details about our experimental setup and the related experimental results.

5.4 Adaptable BCH Encoder

In this section, we propose an adaptable BCH encoder which exploits the shOPPLFSR of Section 5.3. According to the BCH theory, the shOPPLFSR of Fig. 5.6 is a very efficient circuit to perform the computation expressed in Eq. 5.2. However, in the encoding phase,

the message $m(x)$ must be multiplied by x^r before calculating the reminder of the division by $g(x)$ (see Eq. 5.2). This can be obtained without significant modifications of the architecture of shOPPFLSR. It is, in fact, enough to input the bits of the message directly in the most significant bit of the LFSR, instead than starting from least significant bit. Fig. 5.7 shows the high-level architecture of the adaptable encoder.

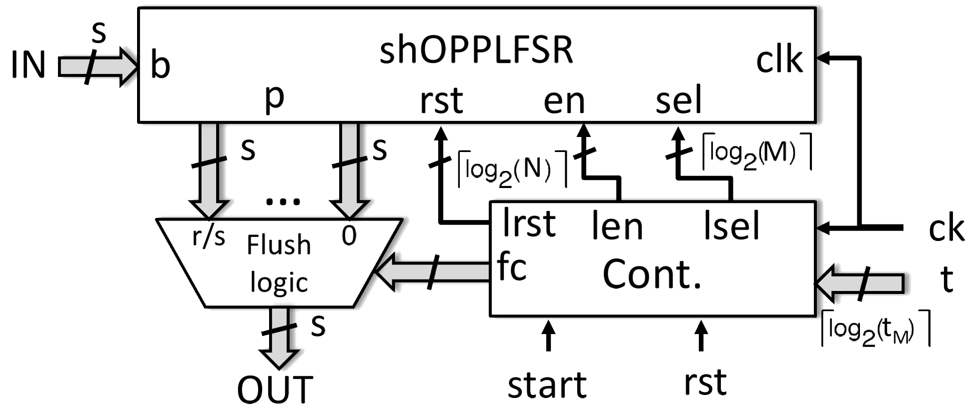


Figure 5.7: High-level architecture of the adaptable encoder highlighting the three main building blocks and their main connections.

The encoder's interface includes: a s-bit input port (`IN`) used to input the k-bit message to encode starting from the most significant bits, a $\lceil \log_2(t_M) \rceil$-bit input port (`t`) selecting the requested correction capability in a range between 1 and t_M, a `start` input signal used to start the encoding process and a s-bit output port (`OUT`) providing the r parity bits. Three blocks compose the encoder: a *shOPPLFSR*, a *flush logic* and a *controller*.

The shOPPLFSR performs the actual parity bits computation. According to the BCH theory, adaptation is achieved by supporting the computation of remainders with t_M generator polynomials, one for each value t may assume. The controller achieves this task in two steps: (i) enabling the proper OPPLFSR through the `len` signal, and (ii) selecting the proper polynomial through the `lsel` signal, according to the desired correction capability `t`. Then, it manages the overall encoding process based on two internal parameters:

1. the number of s-bit words composing the message (fixed at design time)

2. the number of produced s-bit parity words, that depends on the selected correction capability. The flush logic splits the r parity bits into s-bit words, providing them in

output, one per clock cycle.

To further optimize the encoding and the decoding process, since in a flash memory these operations are not required at the same time, the encoder's shOPPLFSR can be merged with the shOPPLFSRs that will be employed in the syndrome computation (see Section 5.5.1), thus allowing additional area saving.

5.5 Adaptable BCH Decoder

Fig. 5.8 presents the high-level architecture of the proposed adaptable decoder. The decoder's interface includes: a $s-$bit input port (`IN`) used to input the $n-$bit codeword to decode (starting from the most significant bits), a $\lceil \log_2(t_M) \rceil -$bit input port (`t`) to select the desired correction capability, a `start` input signal to start the decoding and a set of output ports providing information about detected errors. In particular:

- `deterr` is a $\lceil \log_2(t_M) \rceil -$bit port providing the number of errors that have been detected in a codeword. In case of decoding failure it is set to 0;
- `erradd` and `errmask` provide information about the detected error positions. Assuming the codeword split into $h-$bit words, `erradd` is used as a word address in the codeword and `errmask` is a $h-$bit mask whose asserted bits indicate detected erroneous bits in the addressed word. The parallelism h of the error mask depends on the parallelism of the Chien machine, as explained later in this section;
- `vmask` is asserted whenever a valid error mask is available at the output of the decoder;
- `fail` is asserted whenever an error occurred during the decoding process (e.g., the number of errors is greater than the selected correction capability);
- `end` is asserted when the decoding process is completed.

The full decoder therefore includes four main blocks: (1) the *Adaptable Syndrome Machine*, computing the syndromes of the codeword, (2) the *Adaptable inversion-less Berlekamp Massey (iBM) Machine*, that elaborates the syndromes to produce the error locator polynomial, (3) the *Adaptable Chien Search Machine* in charge of searching for the error positions, and (4) the *Controller* coordinating the overall decoding process.

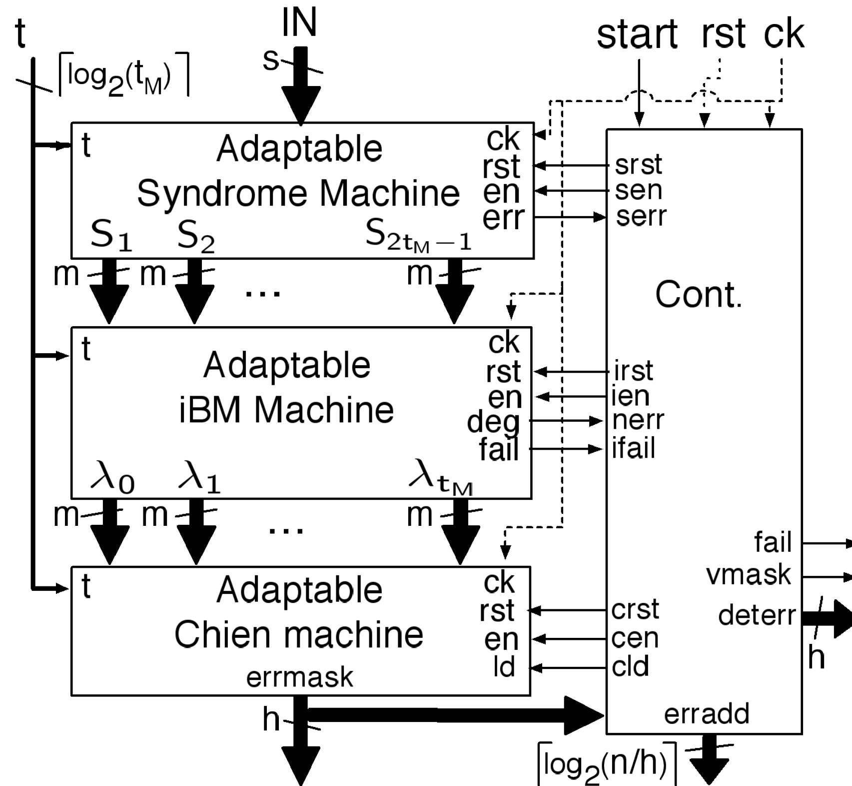


Figure 5.8: High-level architecture of the adaptable decoder, highlighting the four main building blocks: the adaptable syndrome machine, the adaptable iBM machine, the adaptable Chien machine, and the controller in charge of managing the overall decoding process

5.5.1 Adaptable Syndrome Machine

Fig. 5.9 shows the high-level architecture of the proposed adaptable syndrome machine with correction capability $1 \leqslant t \leqslant t_M$.

According to Section 5.2, remainders can be calculated by a set of Parallel LFSRs (PLFSRs) whose architecture is similar to the one of the PPLFSR of Fig. 5.1, with the only difference that the characteristic polynomial is fixed (XOR gates are inserted only where needed, without multiplexers). Each PLFSR computes the remainder of the division of the codeword by a different minimal polynomial $\psi_i(x)$. Given two correction capabilities t_1 and t_2 with $t_1 < t_2 \le t_M$, the set of $2t_1$ minimal polynomials generating the code for t_1 is a subset of those generating the code for t_2. To obtain adaptability of the correction capability in a range between 1 and t_M, the syndrome machine can therefore be designed to compute the maximum number t_M of remainders required to obtain $2t_M$

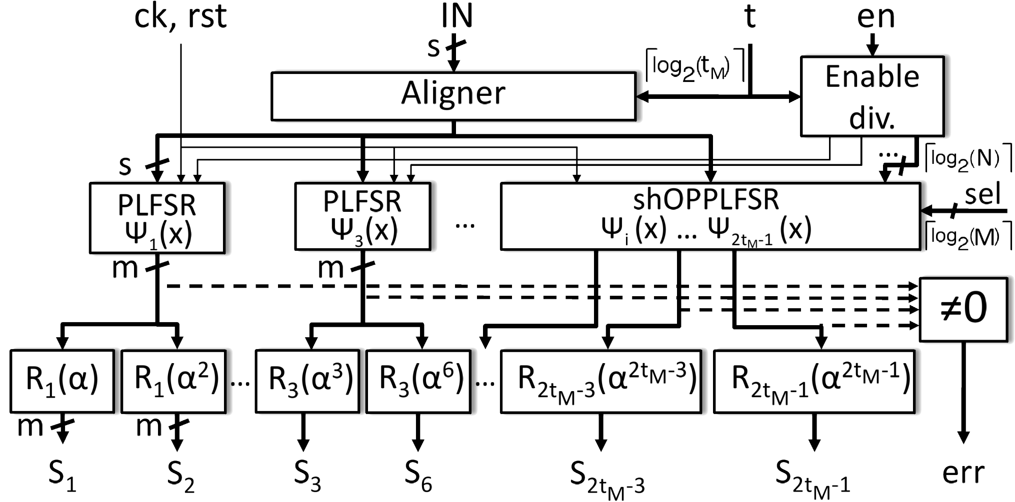


Figure 5.9: Architecture of the adaptable Syndrome Machine

syndromes. Based on the selected correction capability t, only the first t PLFSRs out of the t_M available in the circuit are actually enabled through the *Enable div.* network of Fig. 5.9.

A full parallel syndrome calculator, including t_M PLFSRs, requires a considerable amount of resources that are underutilized in the early stages of the flash lifetime when reduced correction capability is required. To optimize the adaptable syndrome machine and to trade-off between complexity and performance, we exploit the shOPPLFSR introduced in Section 5.2. The architecture proposed in Fig. 5.9 includes two sets of LFSRs for remainder computation: (i) conventional PLFSRs, and (ii) shOPPLFSR. Conventional PLFSRs are exploited for parallel fast computation of low order syndromes required when the requested correction capability is below a given threshold. shOPPLFSR is designed to divide for selected groups of minimal polynomials not covered by the fixed PPLFSRs. It represents a shared resource utilized when the requested correction capability increases. It enables area reduction at the cost of a certain time overhead. The architectural design, chosen for the fixed PLFSRs and the OPPLFSR, enables to trade-off hardware complexity and decoding time, as it will be discussed in Section 5.6.

It is worth to mention here that the parallel architecture of the PLFSR, coupled with the adaptability of the code, introduces a set of additional word alignment problems that must be addressed to correctly adapt the syndrome calculation to different values of t. The syndrome machine receives the codeword in words of s bits, starting from the most

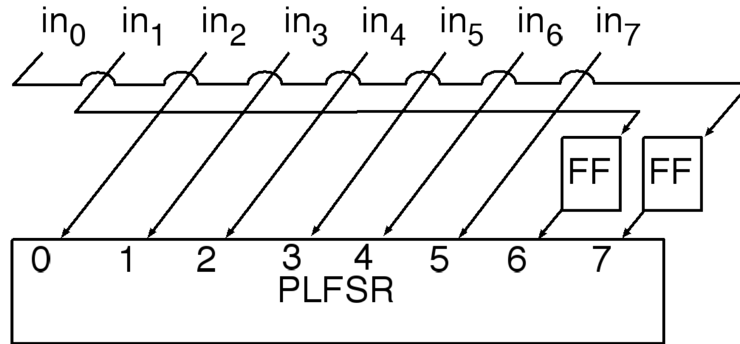


Figure 5.10: Example of the schema of a byte aligner for $t = 2$ and $s = 8$

significant word. When the number of parity bits does not allow to align the codeword to the parallelism s, the unused bits of the last word are filled with 0. To correctly compute each syndrome, the parity bit r_0 of the codeword must enter the least significant bit of each LFSR. The aligner block of Fig. 5.9 assures this condition by properly right-shifting the codeword while it is input into the syndrome machine. Let us consider the following example: $k = 2\text{KB}$, $m = 15$, $t = 2$, $s = 8$ and therefore $r = m \cdot t = 30$. Since 30 is not multiple of $s = 8$, the codeword is filled with two zeros and p_0 is saved in position 2 of the last byte of the codeword ($m_{2047}\, m_{2046} \ldots m_1\, m_0\, p_{29}\, p_{28} \ldots p_1\, p_0\, 0\, 0$). In this case the PLFSRs require a 2-bit alignment, implemented by the network of Fig. 5.10. It simply delays the last 2 input bits resorting to two flip-flops, whose initial state has to be zero, and properly rotates the remaining input bits. Changing the correction capability of the decoder changes the number of parity bits of the codeword, and therefore the required alignment. Given the parallelism s of the decoder, a maximum of s alignments must be provided and implemented in the *Aligner* block of Fig. 5.9.

With the proper alignment, the PLFSRs can perform the correct division and the evaluators can provide the required syndromes. The evaluators are simple combinational networks involving XOR operations, according to the Galois Fields theory (readers may refer to [102] for specific implementation details).

5.5.2 Adaptable Berlekamp Massey Machine

In our adaptable codec, we implemented the inversion-less Berlekamp-Massey (iBM) algorithm proposed in [142] which is able to compute the error locator polynomial $\lambda(x)$ in t iterations.

The main steps of the computation are reported in Alg. 1. At iteration i (rows 2 to 12), the algorithm finds an error locator polynomial $\lambda(x)$ whose coefficients solve the first i equations of (5.3) (row 4). It then tests if the same polynomial solves also $i+1$ equations (row 5). If not, it computes a discrepancy term δ so that $\lambda(x)+\delta$ solves the first $i+1$ equations (row 9). This iterative process is repeated until all equations are solved. If, at the end of the iterations, the computed polynomial has a degree lower than t, it correctly represents the error locator polynomial and its degree represents the number of detected errors; otherwise, the code is unable to correct the given codeword.

Algorithm 1 Inversion-less Berlekamp-Massey alg.

```
1:  λ(x) = 1, k(x) = 1, δ = 1
2:  for i = 0 to t − 1 do
3:      d = Σ_{j=1}^{t} (λ_j · S_{2i−j})
4:      λ(x) = δλ(x) + d · x · k(x)
5:      if d = 0 OR Deg(λ(x)) > i then
6:          k(x) = x^2 · k(x)
7:      else
8:          k(x) = x · k(x)
9:          δ = d
10:     end if
11:     i=i+1
12: end for
13: if Deg(λ(x)) < t then
14:     output λ(x), Deg(λ(x))
15: else
16:     output FAILURE
17: end if
```

The architecture of the iBM machine is intrinsically adaptive as long as one guarantees that the internal buffers and the hardware structures are sized to deal with the worst case design (i.e., $t = t_M$). The coefficients of $\lambda(x)$ are $m-$bit registers whose number depends on the correction capability. In the worst case, up to t_M coefficients must be stored for each polynomial.

The adaptable iBM machine therefore includes two $m-$bit register files with t_M registers to store these coefficients. Whenever the requested correction capability is lower than t_M, some of the registers will remain unused. The number of multiplications performed during the computations also depends on t. Row 3 requires t multiplications, while row 4 requires t multiplications to compute $\delta\lambda_i(x)$ and t multiplications to compute $d \cdot x \cdot k(x)$.

We implemented a serial iBM Machine including 3 multipliers for GF(2^m) to perform multiplications of rows 3 and 4. It can perform each iteration of the iBM algorithm in $2t$ clock cycles (t cycles for row 3 and t cycles for row 4) achieving a time complexity

of $2t^2$ clock cycles. This implementation is a good compromise between performance and hardware complexity. An input t dynamically sets the number of iterations of the algorithm, thus implementing the adaptation.

5.5.3 Adaptable Chien Machine

The overall architecture of the proposed adaptable Chien Machine is shown in the Fig. 5.11. The machine first loads into t_M m-bit registers the coefficients from λ_1 to λ_{t_M} of the error locator polynomial $\lambda(x)$ computed by the iBM machine (`ld` = `0`). The actual search is then started (`ld` = `1`). At each clock cycle, the block performs h parallel evaluations of $\lambda(x)$ in GF(2^m) and outputs a $h-$bit word, denoted as `errmask`. Each bit of `errmask` corresponds to one of the h candidate error locations that have been evaluated. Asserted bits denote detected errors. This mask can then be XORed (outside the Chien Machine) with the related bits of the codeword in order to correct the detected erroneous bits.

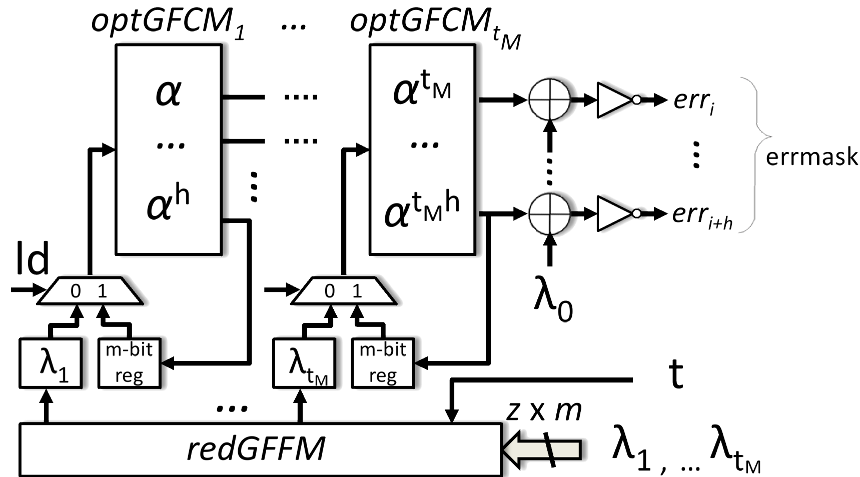


Figure 5.11: Architecture of the proposed parallel adaptable Chien Machine with parallelism equal to h

The architecture of Fig. 5.11 provides an adaptable Chien machine with lower area consumption than other designs [37], having, at the same time, a marginal impact on performance. Four interesting features contribute to such optimization: (i) constant multipliers substructure sharing, (ii) adaptability to the correction capability, (iii) improved fast skipping to reduce the decoding time, and (iv) reduced full GF multipliers area. In the sequel, we briefly address each feature.

The first feature is represented by the optimized GF Constant Multipliers (optGFCM) networks of Fig. 5.11. The h parallel evaluations are based on equation (5.4). In the worst case ($t = t_M$), the parallel evaluation of equation (5.4) requires a matrix of $t_M \times h$ constant Galois multipliers. They multiply the content of the t_M registers by $\alpha, \alpha^2, ..., \alpha^{t_M}$, respectively. However, we can note that each column of constant GF multipliers shares the same multiplicand. Therefore, we can iteratively group their best-matching combinations [39] into the t_M optGFCM networks of Fig. 5.11. Such optGFCMs provide up to 60% reduction of the hardware complexity of the machine with no impact on performance.

The second feature is the adaptability of the Chien machine. The rows of the matrix define the parallelism of the block (i.e., the number of evaluations per clock cycles), while the columns define the maximum correction capability of the block. Whenever the selected correction capability t is lower than t_M, the coefficients of the error locator polynomial of degree greater than t are equal to zero and do not contribute to equation (5.4), thus allowing us to adapt the computation to the different correction capabilities.

The third feature stems from a simple observation. Depending on the selected correction capability t, not all the elements of $GF(2^m)$ represent realistic error locations. In fact, considering a codeword composed of k bits of the original message and $r = m \cdot t$ parity bits, only $k + m \cdot t$ out of 2^m elements of the Galois field represent realistic error locations. Given that an error location L is the inverse of the related GF element ($L = 2^m - 1 - i$), the elements of $GF(2^m)$ in which the error locator polynomial must be evaluated are in the following range:

$$\left[\underbrace{\alpha^{2^m-1}}_{\text{error location L=0}}, \underbrace{\alpha^{2^m-k-m\cdot t}}_{\text{error location } L=k+m\cdot t-1} \right] \tag{5.9}$$

All elements between α^0 and $\alpha^{2^{m-k-m\cdot t}}$ can be skipped to reduce the computation time. Differently from fixed correction capability fast skipping Chien machines, this interval is not constant here but depends on the selected t. The architecture of Fig. 5.11 implements an adaptable fast skipping by initializing the internal registers to the coefficients of the error corrector polynomial multiplied by a proper value $\beta^t_{ini} = \alpha^{2^m-k-m\cdot t-1}$. For each value of t, t_M $m-$bit constant values corresponding to $\beta^t_{ini}, (\beta^t_{ini})^2, \ldots, (\beta^t_{ini})^{t_M}$ must be stored in an internal ROM (not shown in Fig. 5.11) and multiplied by the coefficients λ_i using a full GF multiplier.

This is connected with the last feature, the reduced GF Full Multipliers (redGFFM) network of Fig. 5.11. Each full GF multiplier has a high cost in terms of area. Since they are used only during initialization of the Chien, the redGFFM adopts only $z \leqslant t_M$ full GF multipliers. It also includes a (λ) input port to input z coefficients, per clock cycles, of the error locator polynomial. This network enables to reduce area consumption, at a reasonable cost in terms of latency.

For the sake of brevity, a detailed description of the controller required to fully coordinate the decoder's modules interaction is omitted.

5.6 Experimental Results

This section provides experimental data from the implementation of the adaptable BCH codec proposed on a selected case study.

5.6.1 Automatic generation framework

To cope with the complexity of a manual design of these blocks, a semi-automatic generation tool named ADAGE (ADaptive ECC Automatic GEnerator) [54] able to generate a fully synthesizable adaptable BCH codec core following the proposed architecture has been designed and exploited in this experimentation extending a preliminary framework previously introduced in [28]. The overall architecture of the framework is in Fig. 5.12.

The code analyzer block represents the first computational step required to select the desired code correction capability based on the Bit Error Rate (BER) of a page of the selected flash [107]. The BER is the fraction of erroneous bits of the flash. It is the key factor used to select the correction capability. Two values of BER must be considered. The former is the raw bit error rate (RBER), i.e., the BER before applying the error correction. It is technology/environment dependent and increases with the aging of the page [23, 143]. The latter is the uncorrectable bit error rate (UBER), i.e., the BER after the application of the ECC, which is application dependent. It is computed as the probability of having more than t errors in the codeword (calculated as a binomial distribution of randomly occurred bit errors) divided by the length of the codeword [48]:

$$UBER = \frac{P(E > t)}{n} = \frac{1}{n} \sum_{i=t+1}^{n} \binom{n}{i} \cdot RBER^{i} \cdot (1 - RBER)^{n-i} \tag{5.10}$$

Given the RBER of the flash and the target UBER, Eq. 5.10 can be exploited to compute the maximum required correction capability of the code and consequently the value

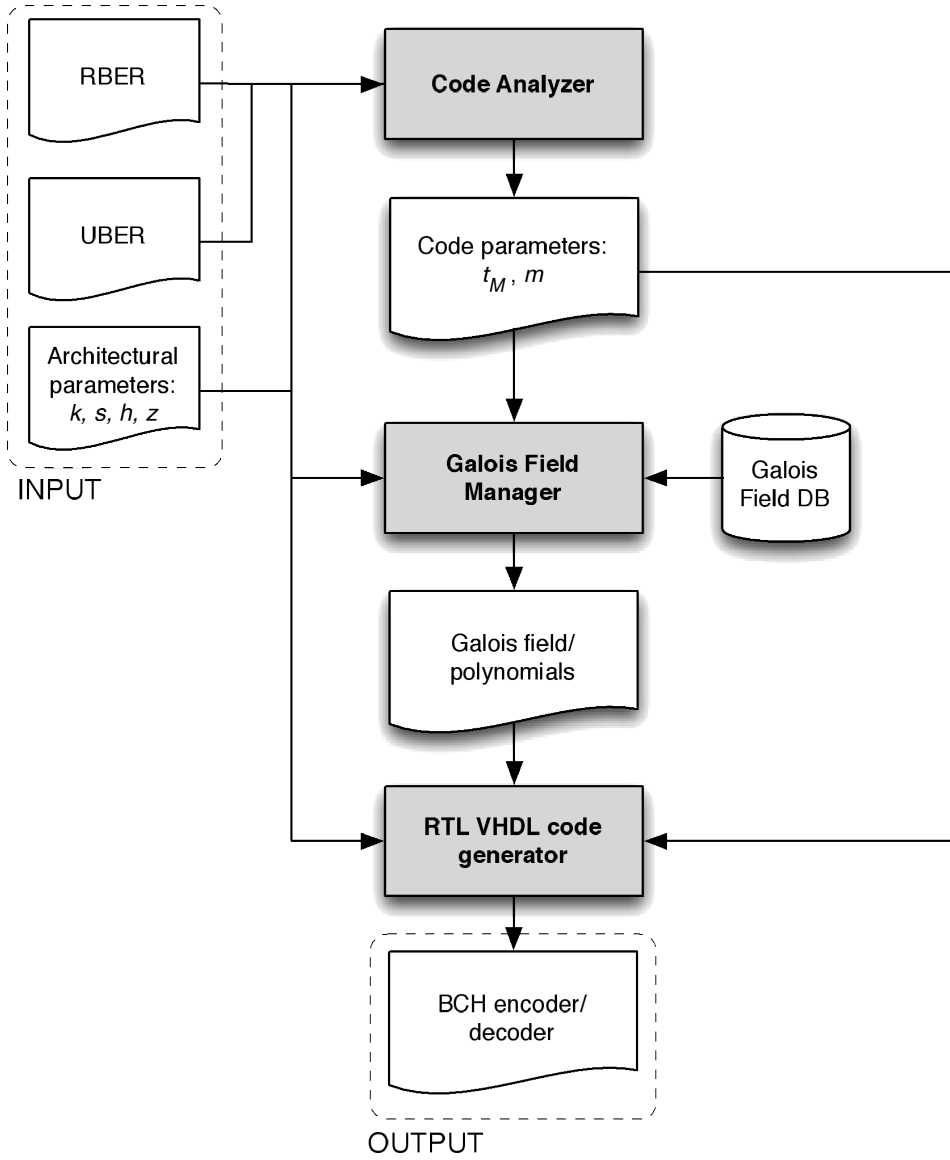


Figure 5.12: BCH codec automatic generation framework.

of m that defines the target GF. Given these two parameters, the Galois Field manager exploits an internal polynomials database to generate the set of minimal polynomials and the related generator polynomials for the selected code.

Finally, the RTL VHDL code generator combines these parameters and generates a RTL description of the BCH encoder and decoder implementing the architecture illustrated in this paper.

The whole framework combines Matlab software modules with custom C programs. The full framework code is available for download at `http://www.testgroup.polito.it` in the Tools section of the website.

5.6.2 Architectural-layer characterization

In our specific design, the ECC sub-system has been implemented to work on a full page of the flash (i.e., $k = 4KB$). We considered a target UBER equal to 1E-11, as in [106]. Based on equation (5.10), Table 5.3 reports the correction capability required to achieve the target UBER considering the RBERs of the various programming algorithms characterized in Chapter 4. Clearly the correction capability required to satisfy the target UBER constraints increases over time. As expected, from the reliability standpoint, the worst performance is provided by the ISPP-RV algorithm. This algorithm requires at the end of the life of the device a correction capability of 450 errors per page. This value would require a considerable hardware and performance overhead that leads to the conclusion that memory pages using the ISPP-RV algorithm will necessarily provide a reduced endurance. For this reason we selected as target maximum correction capability 88 errors per page corresponding to the requirement of the ISPP-SV algorithm at the end of life. Given the selected value of k and t the resulting code is designed over GF(2^{16}) (i.e., $m = 16$).

Table 5.3: Correction capability required by the ECC to achieve a target UBER=1E-11 (Every element of the table reports the memory RBERs for the different programming algorithms (pattern independent) as characterized in Chapter 4, and the needed correction capability).

Alg/progr. cycles	**1**	**100**	**1000**	**10000**	**100000**
ISPP-RV	1.000E-06 / 3	6.104E-05 / 11	3.052E-04 / 28	**1.526E-03 / 88**	9.0332 E-03 / 450
ISPP-SV	1.000E-06 / 3	1.000E-06 / 3	2.747E-04 / 26	3.357E-04/ 29	1.000E-03 / 65
ISPP-DV	1.000E-06 / 3	1.000E-06 / 3	3.052E-05 / 8	3.052E-05 / 8	9.155E-05 / 14

In the remaining of this section the ECC sub-system will be characterized to show the different trade-offs offered by its programmability. It is worth to mention here that

our ECC implementation features a 8-bit parallelism to meet the I/O parallelism of the target flash, and a 8-bit parallelism of the Chien machine allowing 8 evaluations per clock cycle to speed-up the decoding process. Table 5.4 reports the area required for this block synthesized using Synopsys DesignCompiler with the STM-45nm [46] technology library. The full design works at 100MHz clock frequency.

Table 5.4: ECC encoder and decoder area footprint. Synthesis has been performed using the STM-45nm technology library.

	Area (μm^2)
Encoder	169931.76
Decoder	514398.40

Let us start with the evaluation of the amount of redundancy (i.e., parity bits) introduced by the ECC. In the worst case (e.g., $t = 88$) the code requires to store $m \cdot t = 16 \cdot 88 =$ 1408bits. This accounts for about 78.5% of the spare area available on our device that corresponds to 224B per page. ECC parity bits, are not the only extra information stored in a flash memory. High-level functions such as filesystem management and wear-leveling need to save considerable amount of information and when the spare are is not enough a certain amount of pages of the flash must be reserved, thus reducing the overall flash capacity. However, looking at Fig. 5.13, if reduced correction capability is required, either because the device is in the early stage of its life, or because a more reliable programming algorithm is applied, the spare area occupation can be reduced up to 74% (3.57% occupation for $t = 3$). This provides a high degree of freedom for the flash memory controller.

The choice of t also makes it possible to tune the ECC latency and its power consumption.

Fig. 5.14 shows that, carefully tuning the correction capability, the ECC subsystem can introduce a significant save in the decoding time compared to the worst case ($t = 88$). Simulations have been performed in the worst case conditions, i.e., t errors injected into the last bits of the page to make sure that the Chien machine has to evaluate the full page in order to find the corrupted bits. The encoding latency is instead almost constant regardless of the selected correction capability.

Similarly to the ECC latency also the ECC power consumption can be traded-off by carefully selecting the correction capability. Fig. 5.15 shows that, also in this case, we can reach up to ~33% of saving in the decoding power consumption when reducing the correction capability.

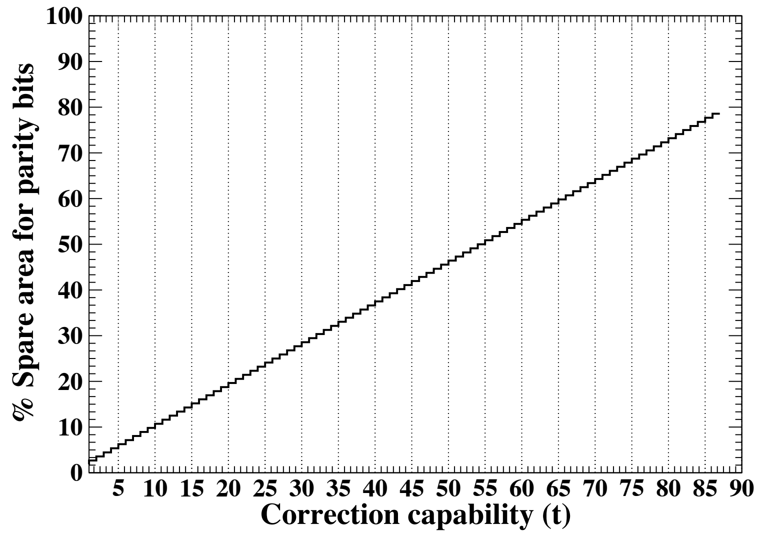


Figure 5.13: Percentage of spare area dedicated for storing parity bits as a function of the selected correction capability.

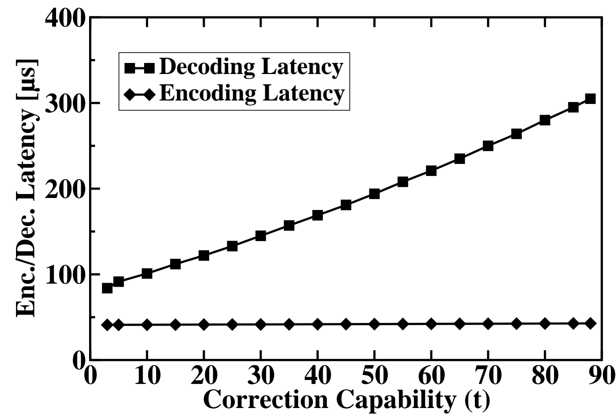


Figure 5.14: Worst case ECC encoding and decoding latency. Simulations have been performed at a clock frequency of 100MHz.

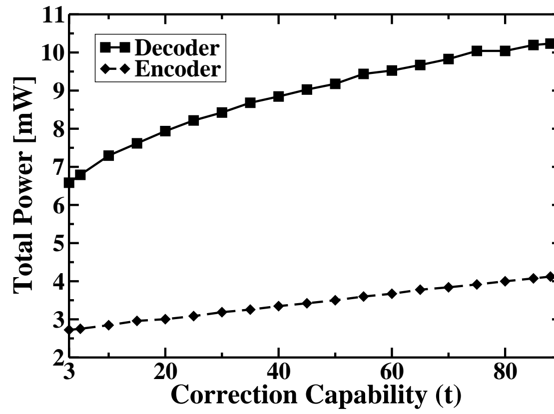


Figure 5.15: Worst case ECC power consumption.

To conclude the characterization of the designed programmable ECC sub-system, Fig. 5.16 reports the relation between UBER and RBER for the selected correction mode obtained by plotting equation (5.10). The figure shows an additional degree of freedom the controller can achieve in which also the UBER can be tuned together with the other parameters.

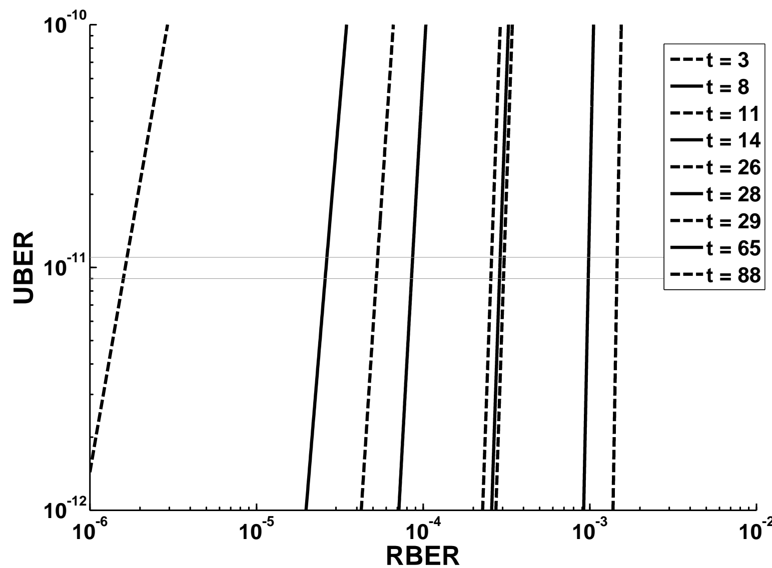


Figure 5.16: RBER vs. UBER relationship for the selected code and selected correction modes.

SUMMARY

This chapter proposed a BCH codec architectures and its related automatic generation framework which enables its code correction capability to be selected in a predefined range of values. Designing an ECC system whose correction capability can be then modified in-field has the potentiality to adapt the correction schema to the reliability requirements the flash encounters during its life-time, thus maximizing performance and reliability.

Experimental results on a selected NAND flash memory architecture proved that the proposed solution reduces spare area usage, decoding time, and power dissipation whenever small correction capability can be selected.

The whole design process was supported by the novel ADaptive ECC Automatic GEnerator (ADAGE) design environment. ADAGE is a fully customizable tool aimed at automatic generation of adaptable BCH architectures. ADAGE is able to automatically generate the VHDL code of the designed adaptable BCH-based architecture. Such a code can be thoroughly simulated, validated and synthesized on ASIC or FPGA.

Table 5.5: Generator polynomial expressed with the corresponding hexadecimal string of coefficients

g_5	0x 0163C68D766635253
g_6	0x 018FBE36E3B716D8BCE32
g_7	0x 01E573FBB06E46A828C1C770C
g_8	0x 01F28E94D9B550543AC42286CF418
g_9	0x 01D6634FC565E6012E441926C07B8D59
g_{10}	0x 018B24C1E935C04DC6BC73E0BDB98405C4EA
g_{11}	0x 01E8B4BA11F717E75A1F5E0EC4FBCD65DA8FFF24
g_{12}	0x 018FB50FA2969CDC5EAFA1C24BD9E5AA92A2227EC668
g_{13}	0x 012E919C715C15310DA7103C0AB656C7FE330613197631D
g_{14}	0x 01E59154D4757E35CBDCE8247F4686EACC2C96C8209D848BDCE
g_{15}	0x 01E12C4539A437988318B8B0A756426E93CD5001031DCB5DC430A0C
g_{16}	0x 01BE62D0F7C4D16FCCDD3CE20D7998280B591702D452F3541A51DA955D8
g_{17}	0x 019755B57BEBA0DD4C284FE4B4F4549C194CA6E75E542322123EAB270447821712
g_{18}	0x 016240D5F338473A9653892D4DDC334AEF9FE78E9B835C10D1C9106B14AA4AB4BD5CD4
g_{19}	0x 01B54AFF801C5EBB55EA214ADCCB051347A16418268264264299431B25E5B7CE34F402D938
g_{20}	0x 01CA788668B1303E48C4A41BE62900685C4A42DB04E267A642AC82884176194501F076D19CF53
g_{21}	0x 015E830624B4D708788177787CA2DC6C89F7558E799E84DD1027034F4DEC7476ADA565B11240FB4EE
g_{22}	0x 01D6ECB0041A40258ADA46542DB3657CFA042227D7CAADD770809ACC680C2886C0EACDC8D81D34565F7FC
g_{23}	0x 0102924C5CEA2B43968EFF54D1E0FAB54DEBFDC54428EDAE6FE2EE724B79CBC072C19CEB766864091E5551A38
g_{24}	0x 0141AE126215097403F13F41BE936020FAA0D6D486AD40BE0BED62DC87C4D8CF945A4D2A804411217E82829127AD

Chapter 6

Cross-layer optimization framework

Contents of this chapter

6.1 EF^{3}S Framework
6.2 Cross-layer Optimized NAND flash access modes
6.3 Storage services at work

The goal of this thesis is to enhance the degree of run-time reconfigurability of an MLC NAND Flash controller through the provision of user-selectable differentiated memory access modes based on an adaptive ECC decoding structure (architecture-level adaptivity) combined with an adaptable memory programming circuitry (physical layer adaptivity).

After having considered the flexibility and the trade-offs in the physical layer and in the ECC sub-system in isolation, this thesis also aims at characterizing the tuning range achievable with such modes. Acting upon their parameters at the same time, the main purpose is to show unprecedented degrees of adaptivity to application requirements in the reliability/performance/power optimization space, thus identifying a set of differentiated access modes that can be configured in the memory controller.

For this purpose, an extensive modeling, simulation and implementation framework has been designed and implemented: EF^{3}S [56]. EF^{3}S is an an easy-to-use, highly configurable, and modular advanced EDA tool which aims at supporting the design of flash-

based systems. It offers the possibility of modeling: the physical NAND device, the memory controller, the NAND flash driver, the Flash File System (including wear leveling and garbage collection), and the application workloads. This framework enable an accurate quantification of the trade-offs between the quality metrics of NVMs accesses on a set of real-life work-loads and benchmark applications. EF3S has been set up to assess the analog and the digital parts of an MLC NAND flash memory sub-system in an homogeneous 45nm industrial technology substrate.

The chapter is organized as follows: Section 6.1 introduces the developed framework called EF^{3}S. Sections 6.2 and 6.3 explores the trade-offs proposed by the cross-layer optimization in the NAND memory controller and shows the performance of the proposed system on a set of real-life applications, respectively.

6.1 EF^{3}S Framework

EF^{3}S enables designers to evaluate the NVM subsystem characteristics (e.g., power consumption, latency, performances, aging) resorting to both synthetic workloads and real traces automatically extracted from real applications.

EF^{3}S is designed for a Linux based environment featuring a Flash File System (FFS), the Linux Memory Technology Device (MTD), and operating system support for raw flash management, including wear leveling and garbage collection. The whole framework combines Linux scripts, C programs, and Matlab software modules. Figure 6.1 shows the EF^{3}S architecture. EF^{3}S comprises three main modules:

- *System Configurator*: the user interface in charge of collecting different design parameters to properly configure the target NVM subsystem;
- *EF^{3}S daemon*, embedded in the Linux Kernel, in charge to monitor the target application workload profiling flash memory operations. Extracted data are then exploited by the Simulation Aging.
- *Simulation Aging*, that elaborates information provided by the System Configurator and the EF^{3}S daemon, simulates the behavior of the target flash memory, and outputs the desired statistics. They include information about throughput, power consumptions, reliability, and aging of the considered system. To assist users in the design space exploration, Fig. 6.1 provides combined and synoptic views of the EF^{3}S architecture.

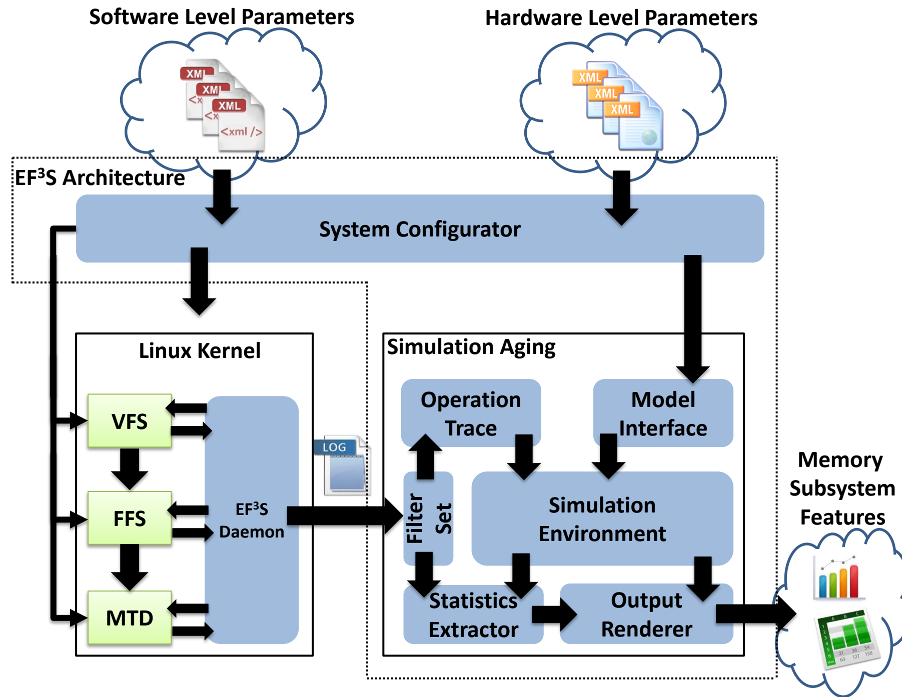


Figure 6.1: EF^{3}S Architecture

In the next section, the different modules composing EF^{3}S will be analyzed in detail.

6.1.1 System Configurator

Designing a NVM subsystem requires the investigation of several design choices, whose characteristics must be specified to EF^{3}S in order to carry out the desired simulations. The *System Configurator* is in charge of collecting these information items. NVM characteristics can be classified into *Hardware Level* and *Software Level* parameters. These design choices fully characterize the NVM subsystem and can be provided in input to the system configuration through a set of dedicated configuration files. The remainder of this section will detail each specific configuration item available in EF^{3}S.

Software Level parameters Software Level design parameters include: the target application and workload, the selected Flash File System, the Wear Leveling algorithm, and the Garbage Collection strategy.

The workload, i.e., the system application interaction with the NVM subsystem, is one of the most critical elements to properly evaluate specific software applications. Two main parameters allow the designer to configure the target workload:

- *Workload Type*. The user may select a set of internally generated workloads produced resorting to the FileBench software suite [7, 134], or resort to a custom workload generated profiling the execution of a custom software application reflecting the specific mission of the simulated system.
- *Workload Running Time*. It is the time for which the workload is executed and profiled. It has to be large enough to perform a significant amount of operations on the system thus collecting enough information about the system's behavior.

In addition, the System Configurator also enables to select among the following system's design choices:

- *Flash File System* (FFS). The selected Flash File System and its configuration parameters strongly affect the NVM performance. EF^{3}S is designed to enable simulations using different Linux based Flash Filesystems including YAFFS2 (Yet Another Flash File System 2), UBIFS (UBI File System) [13] or JFFS2 (Journaling Flash File System 2) [12].
- *Garbage Collection* and *Wear Leveling* algorithms can be switched among a selection of available strategies that depend on the selected filesystem.
- *Caches*. In the NVM Subsystem Software Stack several level of caches at the Virtual File System (VFS) and Flash File System (FFS) level are used. These caches can be enabled or disabled to reflect the desired system configuration.

Software Level parameters are used by the System Configurator to properly instrument the Linux Kernel as well.

Hardware Level parameters At the Hardware Level, EF^{3}S enables the user to configure the NVM subsystem acting on the flash memory, the ECC and the flash interconnection type.

The target flash memory can be configured according to the following parameters:

- *Logical Parameters.* Include page size, block size (or pages per block), device partition size (or blocks per partitions) and total size (or number of device partitions).
- *Operational Parameters*, which may be found on the device datasheet. Such parameters include:
  - device clock frequency;
  - read/program/erase elementary physical operations specifications (e.g., program verify algorithm [144]);
  - elementary operation timing and power consumptions, with the possibility of defining time or aging dependent values;
- *Bare Memory Reliability Model* to be used for reliability status estimation of the memory. Raw Bit Error Rate (RBER) may be assumed as a measure of the reliability status of the bare flash memory as widely presented in Chapter 3.

All provided parameters may be time or memory aging dependent. An accurate formalism has been introduced to allow designers to specify parameters according to either program/erase cycles or time intervals (e.g., flash operation latencies strongly depend on the aging of the memory).

Together with the target flash memory, the ECC subsystem can be also fully configured according to the following parameters:

- *ECC family*, chosen between BCH, LDPC, Reed Solomon (RS) [40] and Product Codes[139];
- *correction capability* or *target reliability*, measured as UBER (Uncorrectable Bit Error Rate) value, i.e., the error rate after applying ECC [107];
- *message length*, with the possibility to apply the code to portions of a page;
- *adaptability*, i.e., a fixed correction capability ECC schema versus a variable one can be chosen [59, 144];
- encoding/decoding *latency and power consumptions* or *implementation technology*, in terms of technology node used to synthesize the ECC, and possibly running clock frequency upper limit.

Finally, the Interconnection System is characterized by the Flash *Interconnection Type* with the rest of the system, i.e., the particular topology of interconnection and protocol.

6.1.2 EF^{3}S daemon

The EF^{3}S daemon interacts with the basic Linux software modules required to deal with flash based storage systems. In particular it filters and records exchanged data between FFS and MTD in order to collect flash memory operation traces.

For sake of comprehension, it is worth mentioning here that the MTD [94] is a driver that provides a uniform interface to different (raw) flash chips. The MTD, if properly instrumented, emulates the presence of a raw NAND flash chip, providing thus the capability to run even when no physical memory devices are available.

In particular, EF^{3}S daemon is instrumented to produce a log file of all flash operations required by the target application workload. This log file is pivotal to Simulation Aging to analyze the system's behavior and provide output statistics.

6.1.3 Simulation Aging

The Simulation Aging is responsible for simulating the configured NVM subsystem.

The *Filter Set* is the main interface of the Simulation Aging with the EF^{3}S daemon. It contains a set of filters, based on regular expressions, which are applied to the log file produced by the EF^{3}S daemon during the execution of the target application. The log file must be properly filtered before performing the required aging simulations according to user requirements. After filtering, a sequence of operations (hereinafter referred to as operation trace) is produced. Each operation is described according to a custom formalism:

$$\langle time \rangle \ \langle r/w\,(/e) \rangle \ \left\langle page\,(/block)\ address \right\rangle$$

where: $\langle time \rangle$ is the time when the operation was issued; $\langle r/w\,(/e) \rangle$ represents the operation type (r stands for read, w for write, e for erase); $\left\langle page\,(/block)\ address \right\rangle$ is the operation target page or block address.

The *Model Interface* acts as an interface between the System Configurator and the Simulation Environment. It elaborates the hardware specific configuration parameters in order to setup the simulation of the computed operation traces. In particular, this module evaluates the piecewise functions defining the Flash Operational Parameters: the right subinterval is identified and expressions are evaluated. The model interface also evaluates the reliability model equations. Finally, the Model Interface module sets the ECC characteristics and correction capability required to satisfy specifications and

computes the ECC run-time parameters such as encoding/decoding latencies and power, resorting to previously characterized ECC schema.

Simulation Environment is the core module of the Simulation Aging. Here all data are combined. The operations to simulate the memory aging are read from the operation trace. Each operation is emulated using the parameters that are concurrently updated by the Model Interface, according to the status (i.e., aging) of the simulated flash memory. At the same time, the Simulator Environment estimates the relevant NVM subsystem features (e.g., average operation latency and throughput, average power and energy per operation, and aging of the pages) that are then provided in output. Of course, produced output accuracy stems from the input models and data accuracy.

The *Statistics Extractor* works in parallel to the Simulation Environment. By solely analyzing the input operation traces, this module extracts statistics about the Flash memory usage, like number of read, write or erase operations per page or block, write/read intensity (ratio between read and write operations) and total number of operations.

Finally, the *Output Renderer* manages the data produced by the Simulator Environment and the Statistics Extractor, to offer a meaningful and synoptic vision of the reliability, performance and behavior of the simulated system by a means of plots, graphs and tables which are automatically generated.

In the next section, EF^{3}S is employed to enable an accurate quantification of the trade-offs between the quality metrics of NVMs accesses on a set of real-life work-loads and benchmark applications, showing the valuable results of the proposed SONVM architecture.

6.2 Cross-layer Optimized NAND flash access modes

The EF^{3}S framework enables an accurate quantification of the trade-offs between the quality metrics of NVMs accesses on a set of real-life work-loads and benchmark applications. Fig. 6.2 provides an overview of how the NAND flash sub-system reacts when selecting different programming algorithms and ECC correction capabilities. Three main parameters of the flash are considered in Fig. 6.2:

- the UBER of the flash;
- the read throughput (RT), i.e., the number of page read requests per second the system is able to serve;

- the write throughput (WT), i.e., the number of write requests per second the system is able to serve.

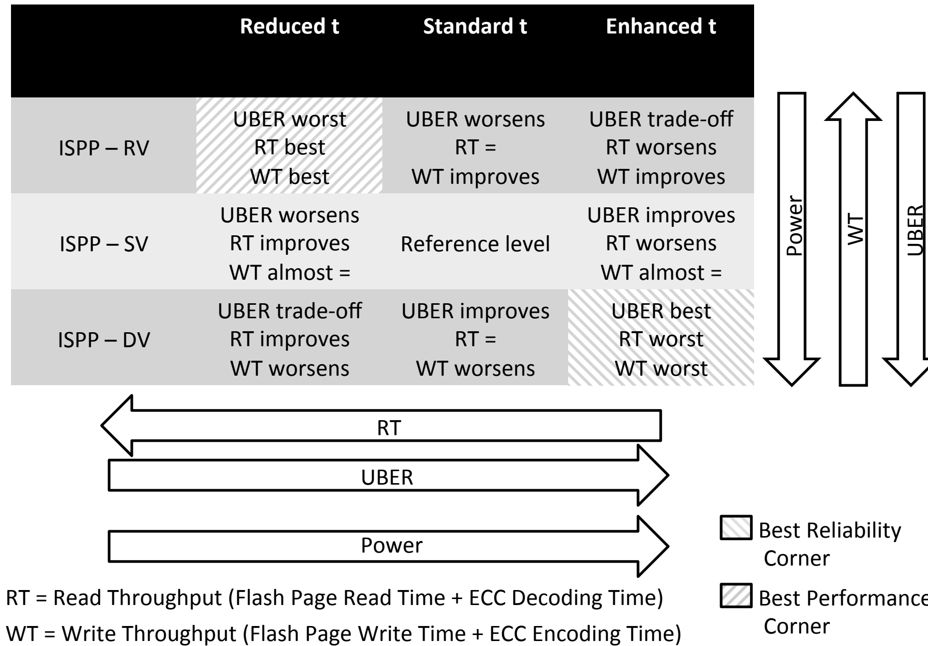


Figure 6.2: Set of access modes provided when tuning the programming algorithm and the ECC correction capability in a cross-layer adaptation framework.

If we consider the ISPP-SV programming algorithm with an ECC designed for UBER of 1E-11 as a reference operating point, the following behaviors can be foreseen:

- UBER worsens when lower values of t, or programming algorithms with reduced verifications are used.
- WT is mainly affected by the programming algorithm. As pointed out in Fig. 5.14 the ECC encoding time is almost constant regardless the selected correction capability.
- RT is mainly affected by the selected ECC correction capability that directly affects the ECC decoding time (see Fig. 5.14). It increases if a lower t is used.
- The combination of reduced t and ISPP-RV represents the best performance corner, but offers the worst reliability.

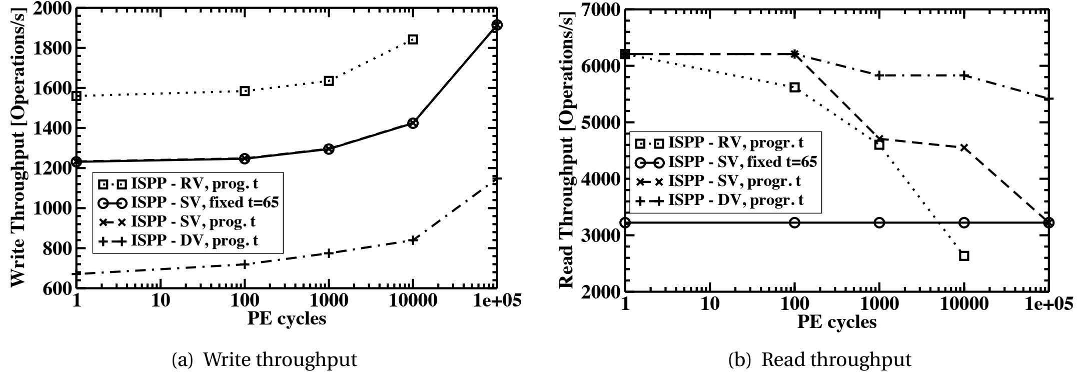
- The combination of increased t and ISPP-DV represents the best reliability corner, but offers the worst performances.
- In the bottom-left and upper-right access modes in the table, the UBER stems from a trade-off between correction strength and the chosen algorithm

An example of these trends can be appreciated in Fig. 6.4. It shows how the modulation effect of RT and WT (for a target UBER=10^{-11}), achievable by changing the programming algorithm selection and ECC correction strength, varies over time along with memory aging. The correction capability of the ECC is adapted as aging increases according to Table 3 to preserve the target UBER in spite of memory aging. For the sake of comparison, the figure shows the performances of a non-adaptive controller using the ISPP-SV programming algorithm and a fixed correction capability $t = 65$ required to meet the target UBER at the end of the memory lifetime.

		PE cycles				
UBER	ISPP alg.	1	10^2	10^3	10^4	10^5
10^{-9}	RV	1	9	24	80	>88
	SV	1	1	22	25	58
	DV	1	1	6	6	11
10^{-11}	RV	3	11	28	88	450
	SV	3	3	26	29	65
	DV	3	3	8	8	14
10^{-13}	RV	4	13	31	>88	>88
	SV	4	4	30	33	70
	DV	4	4	10	10	17

Figure 6.3: Adaptation of the ECC correction capability to the flash aging for different programming algorithms and target UBER

Fig. 6.4 (a) clearly shows that, acting on the programming algorithm, we can modify the write performances of the flash with 30% improvement obtained with *ISPP-RV prog. t* used instead of *ISPP-SV fixed t.* Moreover WT modulation capability is preserved over memory cycling. The non-adaptive and the adaptive ISPP-SV solutions are almost overlapped because encoding latencies are barely affected by the ECC correction strength. Fig. 6.4 (b), instead, shows that we can tune the RT of the system by using ISPP-DV as opposed to ISPP-RV. In these cases the RT can be improved by 36% or degraded by 59% calculated at program/erase (PE) cycle 10k, respectively, and compared to the reference adaptive ISPP-SV solution. Of course, the RT degradation of ISPP-RV is the price to pay for its WT improvement. The comparison of the *ISPP-SV prog. t* curve with the baseline



(a) Write throughput

(b) Read throughput

Figure 6.4: WT and RT comparison among different configurations of the controller for a target UBER=10^{-11}

ISPP-SV fixed t curve shows that tuning the ECC correction capability over the life of the flash enables a significant improvement of the RT with no penalties on WT.

Fig. 6.4 (b) also shows that, in the early stage of the memory life, the modulation capability of RT is marginal. The reason lies in the similar RBER figures of programming algorithms in fresh devices. On one hand this means that RT boost with respect to the reference case will be achieved only after hundreds PE cycles. On the other hand, this also means that in fresh devices WT can be broadly modulated at marginal RT penalty.

Overall, Fig. 6.4 shows a usage model of the access modes: the correction capability is used to preserve a target UBER over flash life, whereas the programming algorithm is used to trade WT with the RT. At a given PE cycle, a higher RT can be achieved by switching the programming algorithm (i.e., from *ISPP-SV prog. t* to *ISPP-DV prog. t*), and the ECC correction strength (since ISPP-DV needs a lower t to preserve the target UBER with respect to ISPP-SV). The WT can be traded-off similarly. Finally, Fig. 6.4 (b) clearly shows that for most of memory life, the non-adaptive approach results in a significant device under-utilization from the RT standpoint.

Other usage models are clearly feasible. For instance, switching from *ISPP-SV prog. t* to *ISPP-DV prog. t*, while keeping t unchanged, results into a minimization of the UBER figure beyond 10^{-11} leaving RT unaltered at the cost of WT. Similarly, a switch to *ISPP-RV progr. t* achieves a WT improvement. If at the same time we decrease t the UBER is largely degraded while improving RT. Otherwise with a constant t the UBER is degraded to lower extent but RT is unaltered.

Finally, the upper-left access mode in Fig. 6.2 can be used in those cases where an ultra-low power operating mode is required while, at the same time, largely degrading UBER and therefore application-perceived reliability are accepted. This could be the case of a mobile system which is about to drain its battery: the user will then decide whether to keep using it in spite of the reduced quality of service or to power it off. In contrast, the lower-right access mode in Fig. 6.2 provides the best achievable reliability at the cost of increased power consumption and largely degraded performance.

Fig. 6.5 summarizes the way UBER can be tuned by selecting different ECC correction capability or programming algorithm. Values in the figure are computed considering the RBER of the flash at 10,000 PE cycles, i.e., quite late in the flash lifetime. Similarly to the performances, Fig. 6.5 shows that we can achieve important trade-offs in the reliability of the access mode, with the possibility of varying the UBER of the NVM system of several orders of magnitude.

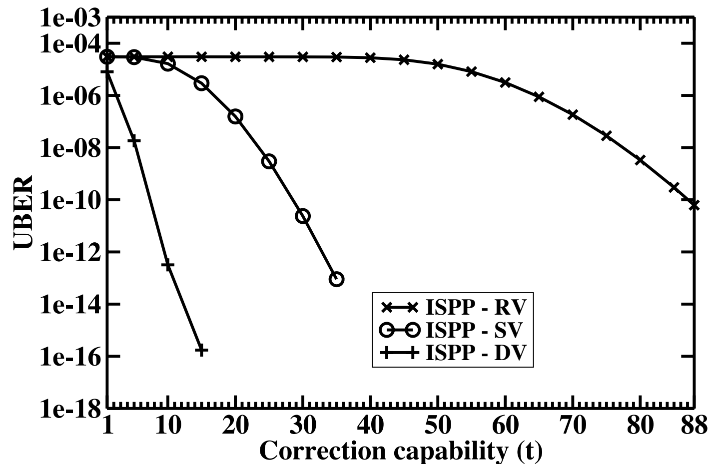


Figure 6.5: Trade-off on the storage reliability by selecting different programming algorithms and different ECC correction capability. UBER is computed at 10,000 PE cycles of the flash.

In order to properly exploit the advantages provided by the proposed adaptive NAND Flash controller, a strategy to decide which memory access mode to use at run-time is of course mandatory. While a complete discussion of this topic is out of the scope of this thesis, a set of preliminary insights can be provided here. There are essentially two factors that must be considered, at run-time, to properly select the available tuning knobs:

1. the application reliability/performance/power requirements and

2. the memory aging.

The first factor is static for a given application or for selected portions of data of an application. Even if not straightforward, applications can be carefully profiled in order to assign different reliability/performance/power requirements to the different set of data they manage. The application profile can be then exploited to choose the best storage service for each type of information. We envision in this thesis to split the flash memory into different partitions providing different services based on Fig. 6.2, according to the requirements of the application to be executed on the target system. Each application can be then instrumented in order to redirect its memory access to the partition providing the access mode that is more suited for the specific data that is going to be accessed. A single application can therefore benefit from data stored in different partitions with different services in order to optimize the overall system performance.

While for a given access mode the selected programming algorithm is in general constant over the memory life-time, the ECC correction capability must be continuously tuned to compensate for the memory aging. Several models in the literature correlate the RBER of a page to the number of performed PE cycles [130]. If this information is constantly tracked during flash operations it can be exploited to adapt the ECC correction capability according to the selected aging model. In this context, one of the most efficient and easy solutions is to demand this operation to the Flash Translation Layer (FTL) or to the File System. At each programming operation the PE cycles of the pages are incremented and stored together with other file system related information. This value can then be exploited at run-time to select the best correction capability every time the page is programmed.

6.3 Storage services at work

To appreciate the benefits of differentiated flash access modes on the execution of a set of real applications, we constructed a simulation environment running under the Linux Operating System. User applications communicate with the Linux Virtual File System (VFS) that decouples the application from the specific file system. YAFFS (Yet Another Flash File System) [15] is the selected flash memory file system. YAFFS is an open-source, fully documented flash memory file system whose source code can be easily modified and instrumented to perform required simulations. Finally, the NAND Driver or Mem-

ory Technology Device (MTD) communicates with the flash memory controller in charge of performing the requested operations on the flash device. The MTD has been instrumented to emulate operations on a NAND flash memory with 4,096 blocks of 128 pages, with a page size of 4 kB. YAFFS has been also instrumented to trace the list of operations performed through the MTD. Read, write and erase operations have been traced. The log essentially contains information about the sequence of operations, the target page address and the timing. To obtain unbiased measurements of the flash performances both VFS and YAFFS caches have been disabled.

Several file system benchmarks are available on the Internet (e.g. IOzone [8], Postmark [81], SPEC benchmarks [11], Filebench [134], etc.). We selected the Filebench benchmark for our analysis. It is an open source File System benchmark originally developed by Sun Microsystem and now by FSL (File systems and Storage Lab) group of the Computer Sciences Department of the Stony Brook University, NY (USA). It provides a large variety of behaviors, also named *personalities*, specified using the Workload Model Language (WML) [7]. They either perform simple file I/O operations, or emulate complex I/O activities.

Among the available personalities, we selected three benchmark applications:

- *varmail*: has different threads performing create-append-sync, read-append-sync, read and delete operations on the files (representing emails) contained in a single directory (workload similar to Postmark);
- *webserver*: opens, reads and closes multiple files in a directory tree while appending data in log file;
- *videoserver*: reads a file set containing videos that are actively served, and writes another file set containing videos that are available but currently inactive.

One of the main characteristic that differentiate the three selected benchmarks is the ratio between the number of read operations ($\#R$) and the number of write operations ($\#W$). This is a critical parameter that influences the type of access mode required by the application to maximize its performance. Table 6.1 summarizes this information. *varmail* is a typical example of write intensive application requiring fast programming of the flash. On the contrary, *videoserver* is a read intensive application requiring fast read access to the data stored in flash. Finally, *webserver* is between the other two benchmarks and performs a more equalized set of read and write operations to the flash.

Table 6.1: $\#R/\#W$ ratios of different Filebench personalities

Personality	$\#R/\#W$
varmail	32.46%
webserver	153.41%
videoserver	1084.70%

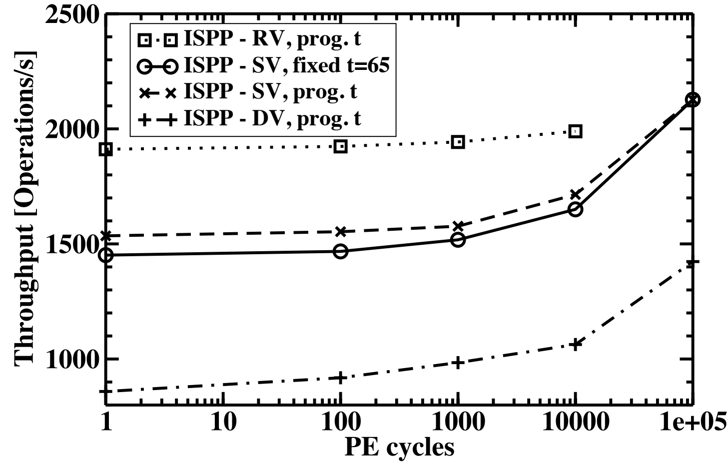


Figure 6.6: Varmail throughput for a fixed UBER=10^{-11}

Fig. 6.6,6.7 and 6.8 show the opportunities the controller programmability provides to the three applications for a target UBER=10^{-11}. All figures report the overall application throughput, i.e., number of operations (read or program operations) performed on the flash per unit of time. Comparison is again performed with a non-adaptive controller using the ISPP-SV programming algorithm and fixed ECC with t=65. Erase operations have been neglected in the calculations of the throughput. In fact, the analysis of the benchmark traces reports that the number of erase operations is far lower than the number of program operations (an average of 1 erase for 2000 program operations).

Looking at Fig. 6.6, that reports the throughput of *varmail*, it is evident that *ISPP-RV prog. t* enables a significant improvement of the overall performances of the application. This improvement comes however with a reduced endurance of the flash due to the high

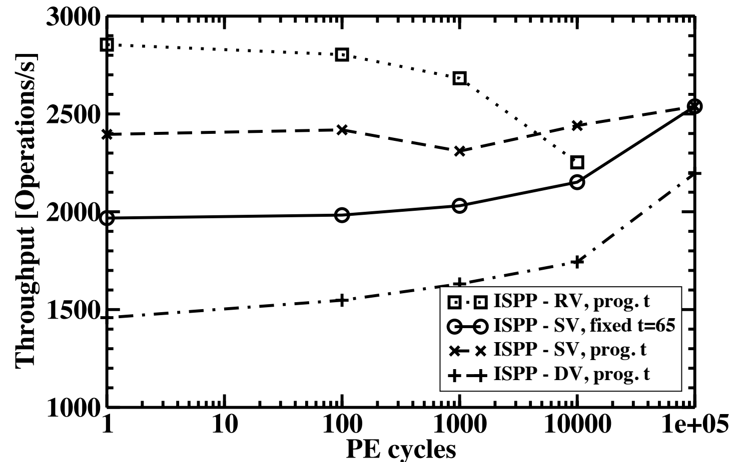


Figure 6.7: Webserver throughput fixed UBER=10^{-11}

RBER introduced by this programming algorithm when the number of PE cycles exceeds 10,000.

If we move, instead, to the opposite application profile represented by the read intensive *videoserver* reported in Fig. 6.8 we can notice an interesting result. Looking at the overall flash lifetime, the *ISPP-SV prog. t* seems the best option for this application even if looking at Fig. 6.4 we could expect better performances from *ISPP-DV prog. t*. The main motivation for this behavior is that the flash programming time is dominant over the flash read time and therefore it negatively influences the overall application performances. This opens new opportunities for the proposed controller. In fact, Fig. 6.8 suggests that not only the ECC correction capability must be adapted to compensate for page aging. In this specific application profile, the ISPP-DV can be selected when the flash reaches more then 10,000 PE cycles to sustain the overall performance and reliability level.

The last situation represented by *webserver* (Fig. 6.7) obviously provides an intermediate behavior. In this situation *ISPP-DV prog. t* reduces the overall performances and is therefore not a good choice for the application. However, both *ISPP-SV prog. t* and *ISPP-RV prog. t* introduce significant performance improvements.

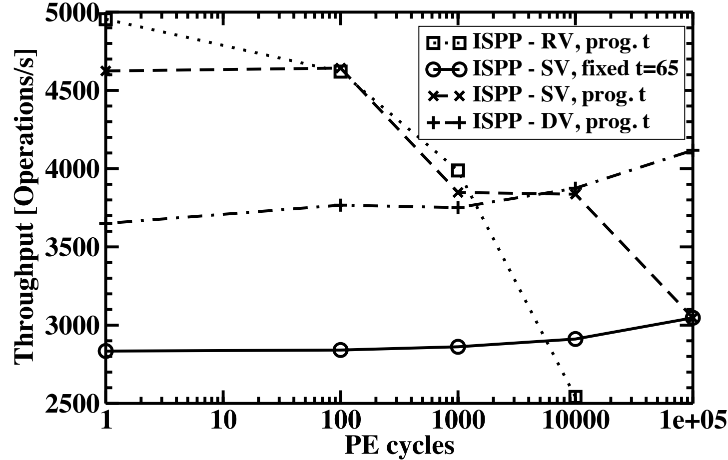


Figure 6.8: Videoserver throughput fixed UBER=10^{-11}

The analysis performed so far highlights how the proposed flash memory controller improves the performance of selected applications when mapped to dedicated access modes. The same programmability can be also exploited to provide access modes with different reliability levels as reported in Fig. 6.9, 6.10, and 6.11 for the *videoserver* application. In this comparison we considered a standard reliability service (UBER=10^{-11}), an enhanced reliability service (UBER=10^{-13}) and a reduced reliability service (UBER=10^{-9}).

Considering the increased reliability service, the target choice will be between *ISPP-SV prog. t* and *ISPP-DV prog. t*. In both cases switching to a higher reliability level does not introduce major penalties in the performances. However, *ISPP-DV prog. t* guarantees performances that are more constant over the overall flash lifetime. This could be a benefit especially when real-time applications are considered. When moving to the reduced reliability service, instead, the choice can be between the *ISPP-RV prog. t* and *ISPP-SV prog. t*. In this case however the choice is a trade-off between performance and memory endurance.

Finally, Fig. 6.12 reports how the reliability of the memory sub-system can now be traded for the reduced power consumption. In power savings scenarios the functionalities of the system need to be preserved in order to either prolong battery life for portable

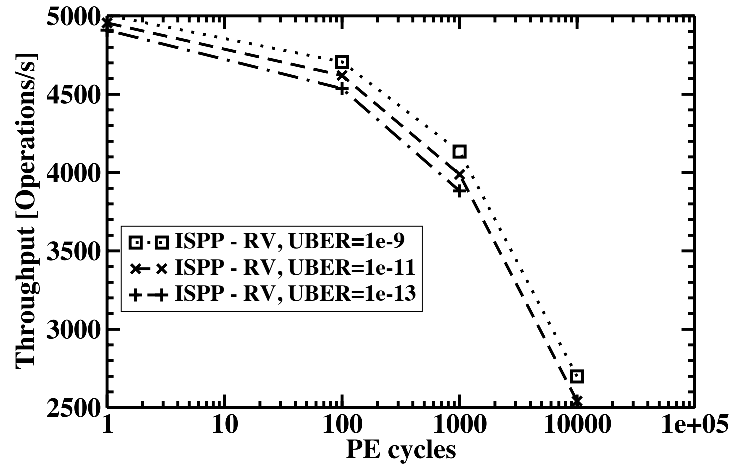


Figure 6.9: Videoserver throughput with ISPP-RV program. *t* at different target UBER

and embedded systems or to reduce cooling issues in high performance computing systems. Under such conditions the quality of service (QoS) of a target application (i.e., video playback) can be degraded to a minimum acceptance level. This is the case of the *ISPP-RV prog. t* access mode, which can significantly reduce the memory energy consumption by a 10% factor at the beginning of the memory lifetime with respect to the non-adaptive ISPP-SV case.

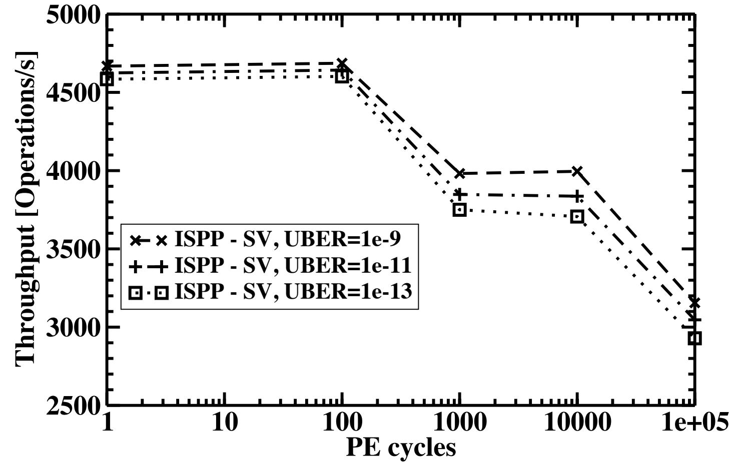


Figure 6.10: Videoserver throughput with ISPP-SV program. *t* at different target UBER

SUMMARY

In this Chapter, we presented benefits due to the valuable interaction between the physical-layer and the architectural-level. The quantification of the trade-offs between the quality metrics of NVM accesses (i.e., reliability, performances, power consumption) have been performed by resorting to a sophisticated framework called EF^3S. Experimental results have been performed on a set of real-life workloads and benchmark applications. We found that a wide range of access modes, each meeting highly differentiated requirements across the embedded and the high-performance computing domains, can be achieved through a cross-layer approach. This opens up new perspectives for a NAND flash device in real-life systems.

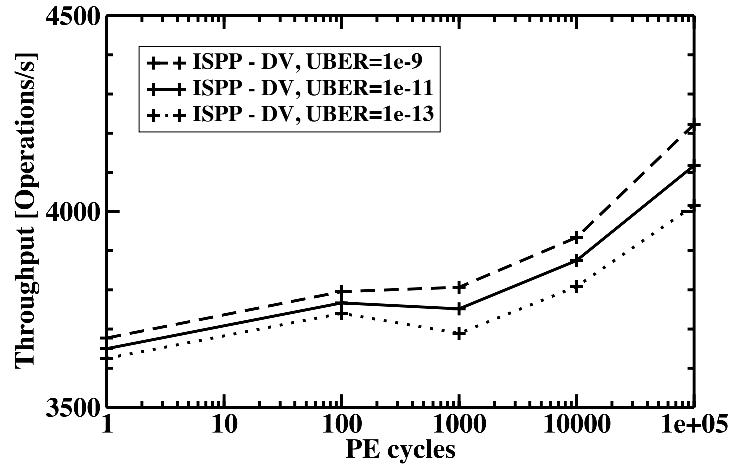


Figure 6.11: Videoserver throughput with ISPP-DV program. t at different target UBER

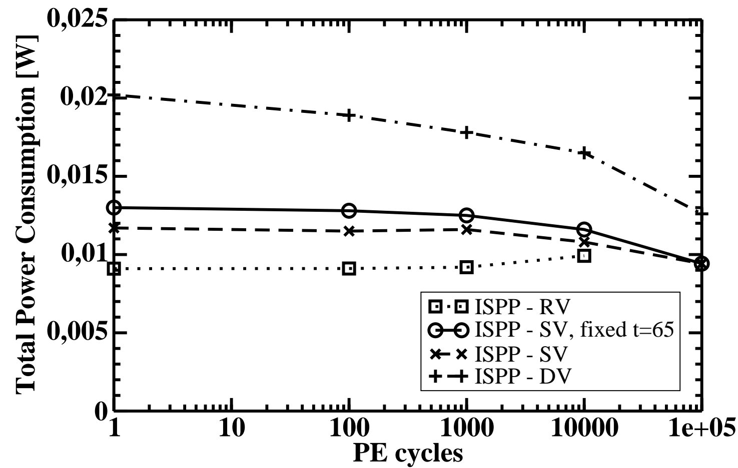


Figure 6.12: Average power per operation during the execution of the videoserver benchmark

CONCLUSIONS

Introducing for the first time the concept of the Service Oriented Non Volatile Memories (SONVMs), the present PhD thesis aims at enhancing the degree of run-time reconfigurability of an MLC NAND Flash controller, through the provision of user-selectable differentiated memory *access modes* (i.e., services). Each mode implements a specific trade-off between read throughput, write throughput, reliability, and power.

The thesis proofs that combining settings at the physical and architectural level in an MLC NAND flash sub-system holds promise of exposing unprecedented trade-offs between performance, reliability and power for memory access. Our cross-layer optimization of NAND flash controllers includes the correction strength of an adaptive ECC framework and the programming algorithm of memory cells, thus yielding access modes for trading off ultra-high performance, and ultra-high reliability. When put at work exploring real-life workloads, the user-selectable access modes prove capable of better adapting to application requirements than non-adaptive controllers. By modeling memory endurance effects, the thesis points out that the most suitable access mode for each application change through the entire memory lifetime, to fully countermeasure the memory aging effects. Finally, the architecture- and circuit level implications of the memory access modes on controller design have been investigated. Based on the gathered results, the RTL coding of the run-time reconfigurable memory controller will be our future work, thus materializing an adaptive NVM sub-system that can complement the current ongoing efforts in adaptive computing.

Furthermore, ongoing research is currently focusing on the exploration of the presented SONVM approach in the framework of the so-called *emerging memories* that are devices featuring both high-performances of SRAMs and DRAMs and the persistence of NVMs at the same time. In this context, we are approaching MRAMs and PRAMs technologies since they are considered the best promising ones.

Appendix A

NAND FLASH MODEL

The flash model developed in this work is a SPICE-based compact model devised for Monte Carlo simulation of a floating gate transistor. The model captures the threshold voltage evolution of a NAND Flash cell during the ISPP algorithm within a memory array, by adding to the calculated cell's threshold voltage, at each time step of the writing algorithm, the following variability sources:

- *Geometrical variability*: since the transistors within the array do not feature the same geometrical parameters, mainly due to lithographic concerns, a displacement on the channel length (L) and channel width (W) from their nominal values σ_L and σ_W is considered in each Monte Carlo run. These latter parameters feature a Gaussian distribution with mean value equal to 1nm and standard deviation of 0.2 nm. Since the geometry of the transistor affects also the threshold voltage evolution, these parameters are calculated before the definition of the transistor structure to be simulated, therefore affecting the final cell's threshold voltage.
- *Cell-to-Cell Coupling*: the SPICE compact model for the NAND array includes parasitic capacitive couplings between each cell and its first neighbors along the same word- and bit-line. The capacitances are derived from 3D-TCAD simulations, and feature the typical values for a 45 nm technology (i.e., roughly about 20 aF). The cell's threshold voltage calculated at each ISPP step takes into account that the elec-

tron tunneling current, and the channel potential of the transistor, deviates from the nominal value by adding a ΔV_{TH} to the voltages exploited for the writing operation.

- *Injection statistics*: the discrete nature of the electronic flow charging the floating gate represents an additional variability source to be considered when dealing with the program operation of nanoscale cells since the statistical process ruling discrete electron injection into the floating gate introduces fluctuations in cell V_{TH} after the application of a writing pulse [128]. On this basis we introduced this additional variability contribution in our compact model for the program operation by adding a displacement from the cell's threshold voltage having the following spread:

$$\sigma_{\Delta V_T} = \sqrt{\frac{q}{\gamma C_{PP}}\left(1 - e^{-\gamma\left(\overline{\Delta_{V_T}}\right)}\right)} \tag{A.1}$$

  where q is the electronic charge, γ is the slope of the tunneling characteristic of the floating gate transistor, C_{pp} is the floating gate capacitance calculated with geometrical variability and $\left(\Delta_{V_T}\right)^{-}$ is the voltage step magnitude of the ISPP algorithm.

- *Random Dopant Fluctuation (RDF)*: The atomistic nature of substrate doping has been clearly shown to result into a fundamental threshold voltage spread for MOS field effect transistors (MOSFETs) given by:

$$\sigma_{RDF} = 3.19 \times 10^{-8} \times \left(\frac{t_{ox}\left(N_A\right)^{0.4}}{\sqrt{WL}}\right) \tag{A.2}$$

  where t_{ox} is the tunnel oxide thickness subjected to geometrical variability and equal to 7.5 nm + 0.1 nm, and N_A is the substrate doping of the cell which follows a profile retrieved by TCAD simulations.

- *Oxide Trap Fluctuation (OTF)*: Referring to traps placed at the substrate/oxide interface (where they have the strongest impact on cell V_{TH}) and assuming a Poissonian fluctuation of their number due to process variability, a spread in cell V_{TH} results according to the following:

$$\sigma_{OTF} = K_{OX} \times t_{ox} \times \frac{\sqrt{Q_{OX}}}{\sqrt{WL}} \tag{A.3}$$

where Q_{ox} is the surface density of traps assumed equal to 10^{-11} cm^{-2} , tox is the tunnel oxide thickness, and K_{ox} is a constant equal to $10^{-6} V \times cm$.

- *Aging effect*: The threshold voltage of a memory cell increases due to charge trapping with the number of write cycles. There are two types of traps that form in the tunnel oxide: interface traps and bulk traps, both of which contribute to the increase in the threshold voltage. It has been shown that both these traps have a power-law relation to the number of cycles on the memory cell [128] as:

$$\Delta Nit = A \times cycle^{0.62} \tag{A.4}$$

$$\Delta Not = B \times cycle^{0.30} \tag{A.5}$$

where A and B are fitting constants, cycle is the number of write cycles on the cell, and the terms ΔN_{it} and ΔN_{ot} are the interface and bulk trap densities respectively. In addition to providing this power-law relationship. The authors calculated the values of constants A and B to be 0.08 and 5, respectively for the considered technology. The total threshold voltage increase due to trapping is divided into interface trap voltage shift (ΔV_{it}) and bulk trap voltage shift (ΔV_{ot}), by using the following equations.

$$\Delta Vit = \frac{\Delta Nit \times q}{C_{ox}} \tag{A.6}$$

$$\Delta Vot = \frac{\Delta Niot \times q}{C_{ox}} \tag{A.7}$$

where C_{ox} is the capacitance of the tunnel oxide.

All these variability sources contributes to the final threshold voltage value approximately with the following percentile values: geometrical variability (15%), oxide trap fluctuations (15%), random dopant fluctuation (25%), parasitic coupling capacitances, injection statistics, and aging (45%).

APPENDIX B

PRINCIPLES OF ERROR CORRECTING CODES

Contents of this appendix

B.1 ECC Principles
B.2 BCH Codes Design Flow
B.3 Error Detecting and Correcting Codes: The actual trend
B.4 Error correcting techniques for future NAND flash memory

This appendix introduces the main concepts related to Error Correcting Code (ECC). The interested reader not familiar with ECCs may delve into the following concepts.

B.1 ECC Principles

The basic principle of all possible ECCs is fairly simple. Let us assume data composed of k-bit. A general ECC algorithm performs two main steps: (i) encoding and (ii) decoding. Fig. B.1 shows the encoding/decoding process.

The encoding process converts (i.e., *encode*) the k-bit data string in a new string (i.e., *codeword*) of n bits, with $n > k$. In other words, $r = n - k$ bits (i.e., *parity bits*) are added to the k-bit data string. The n-bit codeword is stored in the memory and can be affected by errors.

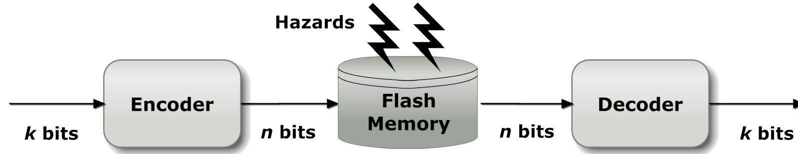


Figure B.1: General Encoding/Decoding structure of Error Correcting Code

The decoding is dual to the encoding process. The n-bit codeword is read out from the memory and is converted (i.e., *decoded*) into a k-bit data string.

Let us summarize the two steps. Encoding adds $r = n - k$ parity bits to the k-bit data string. The codeword is stored in the memory. Decoding converts the n-bit codeword into the most probable k-bit data string. In case of errors, we need suitable metrics to determine them.

Code A *code* is the set of all codewords of a given length that are constructed by adding a specified number of parity bits in a specified way to a specified number of data bits. All the codewords of this set are said to be *valid*, whereas all the others are *not valid*.

Hamming distance The *Hamming distance* of two codewords is the number of corresponding bits that differ between them [64].

Minimum Hamming distance The *minimum Hamming distance* d_{min} of a code is the minimum of the Hamming distance between all possible pairs of codewords of that code. Table B.1 shows a 4-bit binary code with $d_{min} = 2$.

Table B.1: The Hamming distance between pairs of codewords of 4-bit code

	0000	**0011**	**0100**	**0111**	**1000**	**1011**	**1100**	**1111**
0000	-	2	2	2	2	2	2	4
0011	2	-	2	2	2	2	4	2
0100	2	2	-	2	2	4	2	2
0111	2	2	2	-	4	2	2	2
1000	2	2	2	4	-	2	2	2
1011	2	2	4	2	2	-	2	2
1100	2	4	2	2	2	2	-	2
1111	4	2	2	2	2	2	2	-

Table B.1 shows that each (valid) codeword of a code is far *at least* d_{min} from all the other (valid) codewords.

B.1.1 Error Detection

Fig. B.2 shows how a single-bit error can modify a 0000 codeword.

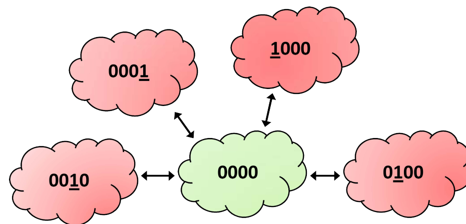


Figure B.2: A "0000" codeword after a single-bit error

E.g., if we read the 0001 codeword from the memory, it is not a valid codeword. In fact, 0001 does not belong to the code of Table B.1. Therefore, the error can be detected.

Fig. B.3 provides a generic example of the encoding/decoding process.

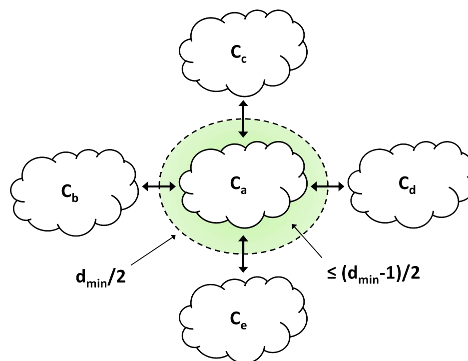


Figure B.3: Generic case Codeword

Fig. B.3 shows that each (valid) codeword is far from the other (valid) codeword at least d_{min}. At least d_{min} single-bit errors have to occur in order to produce another valid codeword. As a consequence, all d_{min}-1 single-bit errors can be detected.

"*A code with d_{min}=d+1 is able to detect d single-bit errors*"

E.g., the code of Table B.1 has d_{min} = 2. Therefore, it is able to detect all 1-single-bit error. In fact, a single-bit error on a valid codeword never provides a valid codeword.

B.1.2 Error Correction

Let us discuss the correction. Supposing a single bit error, Fig. B.4 shows how the wrong 0001 codeword can be corrected.

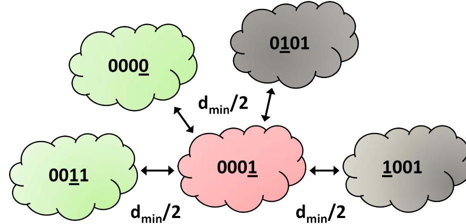


Figure B.4: The wrong "0001" read codeword

0001 is "halfway" between any pair of these codewords[1]. Therefore, it is not possible to understand which codeword 0001 originally was. In other words, this code can only detect 1-single-bit errors and is not able to correct any error.

If the codeword C_a of Fig. B.3 is affected by less than $d_{min}/2$ single bit errors, then the closest codeword to the faulty one is C_a itself.

> "*Any codeword affected by* $\#errors \leq (d_{min} - 1)/2$ *is correctable. Therefore, the correcting power of the code is* $t = \lfloor (d_{min} - 1)/2 \rfloor$"

In order to correct t errors, we need a code with:

$$d_{min} \geq 2t + 1 \tag{B.1}$$

B.1.3 Hamming bound

Let us assume to have a n-bit codeword, a k-bit data, q symbols[2], minimum Hamming distance d_{min} and a correction capability $t = \lfloor (d_{min} - 1)/2 \rfloor$.

Eq. B.2 has to be satisfied in order to proof the validity of Eq. B.1.

$$n - k \geq log_q \left\{ \sum_{i=0}^{t} \left[\binom{n}{i} (q-1)^i \right] \right\} \tag{B.2}$$

We usually refer to Eq. B.2 as *Hamming bound* [63].

[1] "0101" and "1001" are not valid codewords and will be not valid options
[2] if q = 2, symbols are called bits

B.2 Bose-Chaudhuri-Hocquenhem Codes Design Flow

Fig. B.5 resumes the BCH codes design flow.

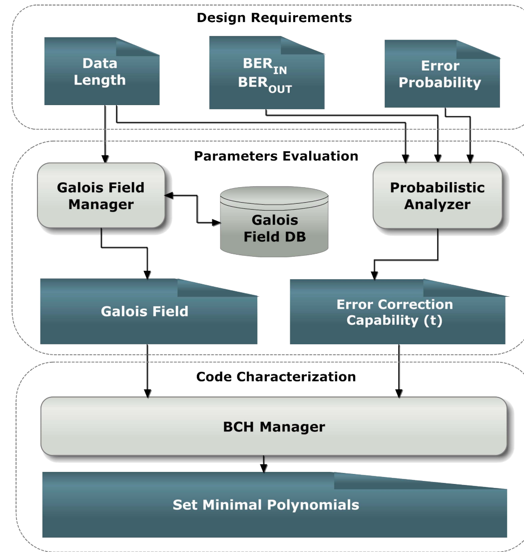


Figure B.5: BCH Code Design Flow

Three main functional steps compose the BCH design flow: (i) *Design Requirements*, (ii) *Parameters Evaluation*, and *Code Characterization*. After the last step, the BCH code is completely defined.

B.2.1 Design Requirements

The first step of each BCH code design flow is to define the mission-critical *requirements*. ECC algorithm works on data of fixed length (i.e., *Data Length*). The correction capability is determined w.r.t. probabilistic studies. The Bit Error Rate (BER) of the page [107], i.e., the fraction of its erroneous bits, is mainly composed by two values: (i) Raw BER (RBER) and (ii) Uncorrected BER (UBER).

The former is the Raw BER (RBER), i.e., the BER before applying the error correction. RBER is technology/environment dependent and is not constant; it increases with aging of the page [23, 107].

The latter is the Uncorrected BER (UBER), i.e., the BER after the application of the ECC, which is application dependent. It can be computed as the probability of having

more than t errors in the codeword (calculated as a binomial distribution of randomly occurred bit errors) divided by the length of the codeword [48]:

$$UBER = \frac{P(E > t)}{n} = \frac{1}{n} \sum_{i=t+1}^{n} \binom{n}{i} \cdot RBER^{i} \cdot (1 - RBER)^{n-i} \tag{B.3}$$

if $n \cdot RBER \ll 1$, [63] rewrites Eq. B.3 as:

$$UBER \approx \frac{1}{n} \cdot \binom{n}{t+1} \cdot RBER^{t+1} \cdot (1 - RBER)^{n-t-1} \tag{B.4}$$

B.2.2 Parameters Evaluation

The Bit Error Rate (BER) of the page [107], i.e., RBER and UBER, is the key factor used to select the correction capability. Fig. B.6 shows the resulting UBER for $k = 2^{14} = 16,384 bits = 2 Kbytes$ and $t = \{0, 1, 5, 10, 15\}$.

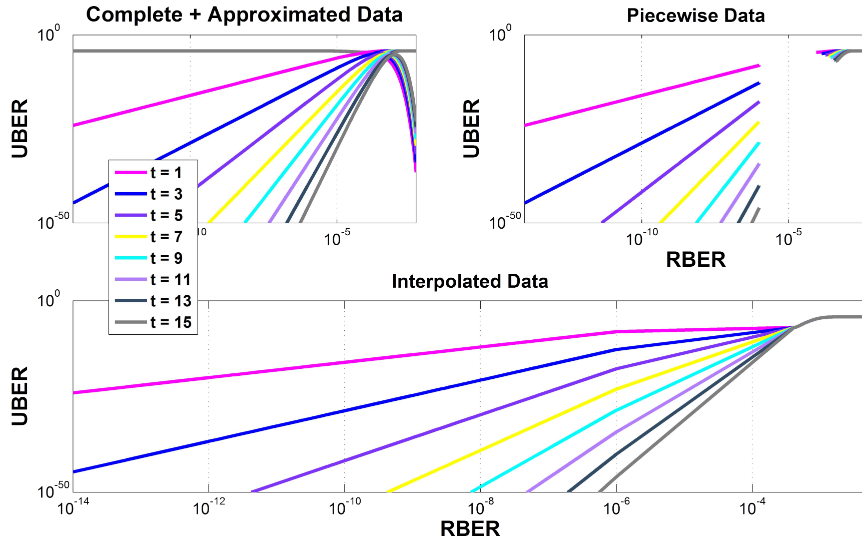


Figure B.6: Examples of Raw BER and Uncorrected BER

The second parameter is the Galois Field (GF). Many codes are based on the abstract algebra and, in particular, on GF [14]. A GF is a finite field with order q, i.e., it has a finite number of elements represented with q symbols). The set of m-tuples of elements from GF is the GF(q^m) vector space. Linear q-ary are a set of m-tuples over GF(q) or, in other words, are subspaces of GF(q^m) [63]. A GF(q^m):

- contains q^m elements, defined as $p_m(x=\alpha)=0 \Longleftrightarrow \alpha^m = b_{m-1}a^{m-1} + b_{m-2}a^{m-2} + \ldots + b_0$;
- all elements can be expressed as α^i with $i\epsilon\left(0,\ldots,q^m-2\right)$;
- always $\alpha^{q^m-1} = 1 = \alpha^0$;
- is closed with respect to addition and multiplication (i.e., the sum or the product of two codewords is a codeword);

Different GFs matches different codes. In particular, two main parameters set the GF: (i) the data length *k* and (ii) the correction capability *t*.

E.g., if q = 2, Eq. B.5 set the minimum GF(2^m) required for the data length *k* [102].

$$k + m \times t \leq 2^m - 1 \tag{B.5}$$

E.g., replacing k = 2^{14} = 16,384 bits = 2KBytes into Eq. B.5, we need at least a Galois Field with $2^m = 2^{15}$ = 32,767 elements.

Spare area and parity bits Eq. B.5 set the minimum m to generate the related GF^m. The number of parity bits is denoted as $r = m \times t$. Such r parity bits are usually stored in the spare area of the flash memory. Therefore, a proper trade-off is needed when designing the ECC in terms of resources overhead.

B.2.3 Code Characterization

Finally, we exploit the correcting power t and the Galois Field to generate the *Minimal Polynomials* $\psi_1(x)$, $\psi_2(x)$, ..., $\psi_{2t}(x)$ [14, 102]. They fully characterize the BCH code.

The set of *Minimal Polynomials* defines the *Polynomial Generator g(x)* of the BCH code [14] as:

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

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

Table B.2 summarize the main BCH code properties.

Table B.2: 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

B.2.4 Shortened Codes

In system design, a code of suitable natural length or suitable number of information digits usually cannot be found. Therefore, it may be desirable to *shorten* a code to meet the requirements. Whenever $n = k + r < 2^m - 1$, the BCH code is called *shortened* or *polynomial*. In a shortened BCH code the codeword includes less binary symbols than the ones the selected Galois field would allow. The missing information symbols are imagined to be at the beginning of the codeword and are considered to be 0. A shortened code has at least the same error-correcting capability as the code from which it is derived [89].

E.g., protecting k = 2^{14} = 16,384 bits data length implies to adopt a GF with 32,767 elements (refer to Eq. B.5). Assuming to correct $t = 5$ errors, we have a resulting codeword n = k+m×t = 16,384 + 15×5 = 16,459 bits < 32,767 = 2^{15}-1. Therefore, we may adopt a code which is shortened of 32,767 - 16,459 = 16,308 bits. A complete BCH[n, k, t] = [32,768, 16,384, 5] becomes a shortened BCH[16,459, 16,384, 5] BCH code.

B.3 Error Detecting and Correcting Codes: The actual trend

ECCs are moving toward two main directions [52]: (i) stronger ECCs and (ii) larger data block.

A stronger ECC has higher correcting power t. However, bigger t implies a higher number $r = m \times t$ of check bits. An higher complexity is also required to detect/correct higher number of errors.

On the other hand, the current trend is to adopt k = 512 Byte. A bigger data length size k may better handle higher concentration of errors. However, bigger k implies bigger symbol size (see Eq. B.5).

Fig. B.7 shows an example of moving toward bigger data length.

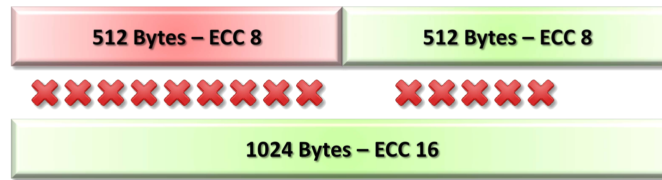


Figure B.7: ECC Example for point "Large Block..."

The first part of Fig. B.7 has two data blocks with k = 512 Bytes. Each block is protected with an ECC with t = 8. This is usually denoted as ECC-8. The second part of Fig. B.7 has one block with k = 1,024 Bytes with ECC-16.

Although the situation looks similar, having 9 and 5 errors in the two k = 512 Bytes block implies a critical failure. Having 9 + 5 = 16 errors are correctable within the k = 1,024 Bytes data blocks.

B.3.1 Examples

Fig. B.8 shows the UBER for several ECCs.

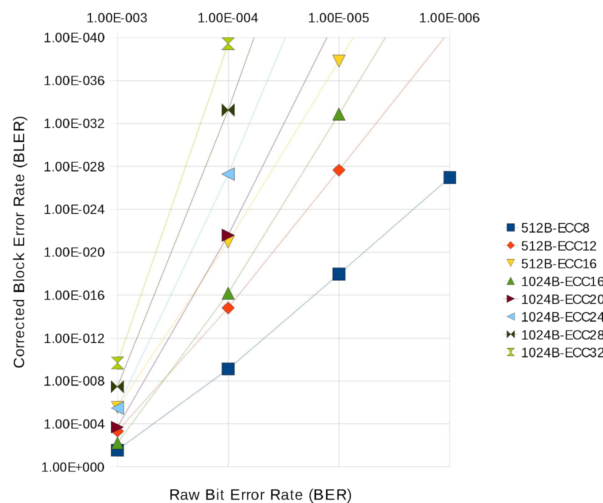


Figure B.8: Uncorrected BER for different ECCs

Fig. B.8 shows that moving toward bigger data blocks improves the UBER. Furthermore, a 512B-ECC16 and a 1024B-ECC16 are equivalent from a UBER standpoint. We provide some simple examples to understand the trade-off to tackle during ECC design.

Example 1 Fig. B.9 shows a first possible example.

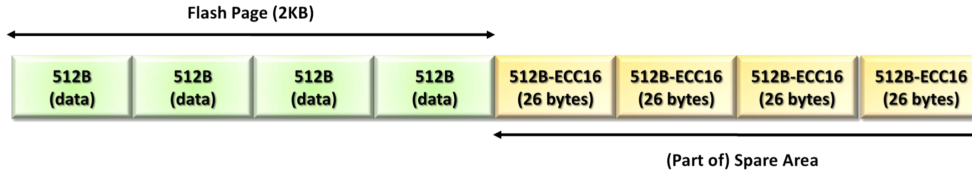


Figure B.9: 512B-ECC16 protecting a 2KB page

Let us assume k = 512 Bytes protected by ECC16 (i.e., 16 errors can be corrected). This is usually denoted as 512B-ECC16. We need:

- **Parity Symbol Size** (m): Eq. B.5 set $m = 13$, i.e., 13-bit parity symbols;
- **Correcting Power**(t): $t = 16$, which implies 13 bit × 16 parity symbols/block = 26 Bytes/block;
- **Complexity**: a 512B-ECC16 requires 4×26 Bytes = 104Byte;

Example 2 Fig. B.10 shows another example.

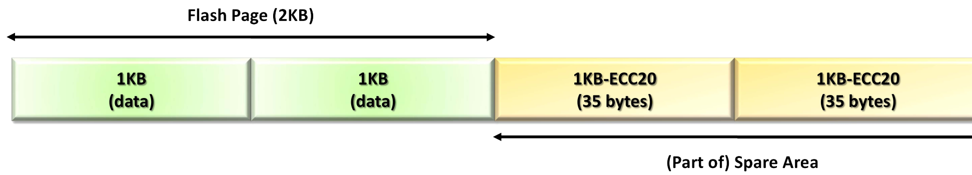


Figure B.10: 1KB-ECC16 protecting a 2KB page

Let us assume k = 1 KBytes protected by ECC20 (i.e., 20 errors can be corrected). This is usually denoted as 1KB-ECC20. We need:

- **Parity Symbol Size** (m): Eq. B.5 set $m = 14$, i.e., 14-bit parity symbols;
- **Correcting Power**(t): $t = 20$, which implies 14 bit × 20 parity symbols/block = 35 Bytes/block;
- **Complexity**: a 1KB-ECC20 protecting a 2KB page requires 2×35 Bytes = 70Byte/-page;

As well as Fig. B.8 shows, the 1KB-ECC20 (Fig. B.9) provides a better UBER than 512B-ECC16 (Fig. B.10), but at lower resource overhead in terms of occupied spare area.

B.4 Error correcting techniques for future NAND flash memory

Thanks to their lower RBER, a 512B-ECC1 (i.e., single-bit correction) may be sufficient for Single Level Cell (SLC) NAND flash. Multi Level Cell (MLC) NAND flashes have higher RBER. Therefore, they require higher correction capability (e.g., at least 512B-ECC4) [57].

20nm NAND flash The continuous scaling-down and the related increasing density of NAND flash implies to adopt proper ECC controllers and algorithms. The first 20nm NAND flash devices are currently available [105]. Such a quick scaling-down implies fewer electrons to enter the Floating Gate (FG). Therefore, there is a higher uncertainty about the charge in the FG.

More bits per cell Nowadays, MLC-based NAND flash can store up to 4 or 8 bit per cell. Although the density of the memory is dramatically increased, also the possible disturbances are much worse. As a consequence, ECCs have to increase their correcting power.

Larger page size The current trend is to increase the page size. 4KB or also 8KB is the most common page size, especially for Solid State Drive.

APPENDIX C

LIST OF SYMBOLS AND ACRONYMS

Due to the large number of symbols used in this thesis to support the description of covered material, we provide the following list of symbols and abbreviations. This list is intended to help the reader identify the meaning of a given symbol or acronym in a fast and easy way.

ADAGE — ADaptive ECC Automatic GEnerator

B — Bulk

BC — BL Coupling

BCH — Bose-Chaudhuri-Hocquenhem

BED — Bit-line Erase Disturbance

BL — Bit-Line

BED — Bit-line Erase Disturbance

BER — Bit Error Rate

BPD — Bit-line Program Disturbance

CC — Capacitive Coupling

CFAC — Coupling Fault between Adjacent Cells

CG	Control Gate
D	Drain
DC	Direct Coupling or Direct field effects
DDR	Double Data Rate
DRAM	Dynamic RAM
ECC	Error Correcting Code
FFS	Flash File System
FG	Floating Gate
FN	Fowler-Nordheim
FTL	Flash Translation Layer
GF	Galois Field
HD	Hard Disk
ISPP	Incremental Step Pulse Programming
MLC	Multi Level Cell
NOP	Number Of PPP
NVM	Non Volatile Memory
OED	Over-Erase Disturbance
OEP	Over-Erase Program
OPD	Over-Program Disturbance
OS	Operating System
PD	Program Disturbance
PPP	Partial Page Programming
PCB	Printed Circuit Board

RAM	Random Access Memory
RD	Read Disturbance
RBER	Raw BER
RDA(E)	RD Addressed Erase
RDA(P)	RD Addressed Program
RDU(E)	RD Unaddressed Erase
RDU(P)	RD Unaddressed Program
ROM	Read-Only Memory
RS	Reed-Solomon
S	Source
SAF	Stuck-At Fault
SG	Select Gate
S	Source
SILC	Stress Induced Leakage Current
SLC	Single Level Cell
SONVM	Service-Oriented Non Volatile Memory
SRAM	Static RAM
SSI	Source Synchronous Interface
SSD	Solid State Drive
UBER	Uncorrected BER
WED	Word-line Erase Disturbance
WL	Word-Line
WPD	Word-line Program Disturbance

BIBLIOGRAPHY

[1] Micron mt29f16g08ababa nand flash datasheet, 2009.

[2] *SLC vs MLC An analysis of Flash memory*. Available at www.supertalent.com/datasheets/SLC_vs_MLC%20whitepaper.pdf, 2009.

[3] Conmpact flash specification. Retrieved August 24, 2013 from the World Wide Web www.compactflash.org, 2010.

[4] Sd cards specification. Retrieved August 24, 2013 from the World Wide Web www.sdcard.com, 2010.

[5] Barco single-port ahb nand flash controller ba315a datasheet. web available resource - http://www.barco.com/projection_systems/downloads/BA315A_FS.pdf, 2012.

[6] Evatronix nandflash-ctrl nand flash memory controller. web available resource - http://www.evatronix.pl/products/docs.html?id=10&product=TkFOREZMQVNILUNUUkw=, 2012.

[7] Filebench. web available resource - http://sourceforge.net/apps/mediawiki/filebench/index.php?title=Filebench, 2012.

[8] IOzone file system benchmark. web available resource - www.iozone.org, 2012.

[9] Open nand flash interface. web available resource - http://onfi.org/, 2012.

[10] Samsung kfg4gh6x4m 4gb flex-onenand m-die datasheet, 2012.

[11] Spec standard performance evaluation corporation. web available resource - http://www.spec.org, 2012.

[12] JFFS: The Journalling Flash File System. Accessed, Feb 2013.

[13] UBIFS - UBI File-System. Accessed, Feb 2013.

[14] Jiri 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, 1st edition, 1991.

[15] Aleph One Ltd. Yet another flash file system 2 (YAFFS2). Retrieved February 24, 2013 from the World Wide Web `http://www.yaffs.net/`, 2011.

[16] G. Atwood, A. Fazio, D. Mills, and B. Reaves. Intel strataflash memory technology overview. *Intel Technol. J*, vol. 1, 1998.

[17] Sung-Ho Bae, Jeong-Hyun Lee, Hyuck-In Kwon, Jung-Ryul Ahn, Jae-Chul Om, Chan Hyeong Park, and Jong-Ho Lee. The 1/f noise and random telegraph noise characteristics in floating-gate nand flash memories. *IEEE Transactions on Electron Devices*, vol. 56, num. 8, pp. 1624–1630, 2009.

[18] E. Berlekamp. Goppa codes. *IEEE Transactions on Information Theory*, 19, 5, 590 – 592, sep 1973.

[19] Elwyn R. Berlekamp. *Algebraic coding theory*. McGraw-Hill, 1968.

[20] D. Bertozzi, S. Di Carlo, S. Galfano, M. Indaco, P. Olivo, P. Prinetto, and C. Zambelli. Performance and reliability analysis of cross-layer optimizations of NAND flash controllers. *Submitted to ACM Transactions on Embedded Computing Systems*, 2013.

[21] R. Bez, E. Camerlenghi, A. Modelli, and A. Visconti. Introduction to flash memory. *Proceedings of the IEEE*, vol. 91, num. 4, pp. 489–502, 2003.

[22] R. C. Bose and D. K. Ray-Chaudhuri. On a class of error correcting binary group codes. *Information and Control*, 3, 1, 68–79, 1960.

[23] Joe Brewer and Manzur Gill. *Nonvolatile Memory Technologies with Emphasis on Flash: A Comprehensive Guide to Understanding and Using Flash Memory Devices*. Wiley-IEEE Press, January 2008.

[24] Dae-Seok Byeon, Sung-Soo Lee, Young-Ho Lim, Jin-Sung Park, Wook-Kee Han, Pan-Suk Kwak, Dong-Hwan Kim, Dong-Hyuk Chae, Seung-Hyun Moon, Seung-Jae Lee, Hyun-Chul Cho, Jung-Woo Lee, Moo-Sung Kim, Joon-Sung Yang, Young-Woo Park, Duk-Won Bae, Jung-Dal Choi, Sung-Hoi Hur, and Kang-Deog Suh. An 8 gb

multi-level nand flash memory with 63 nm sti cmos process technology. *Solid-State Circuits Conference, 2005. Digest of Technical Papers. ISSCC. 2005 IEEE International*, pp. 46–47 Vol. 1, 2005.

[25] P. Cappelletti. *Flash memories.* Springer, 1999.

[26] P. Cappelletti, C. Golla, P. Olivo, and E. Zanoni. *Flash Memories.* Kluwer, 1999.

[27] M. Caramia, S. Di Carlo, M. Fabiano, and P. Prinetto. Flash-memories in space applications: Trends and challenges. *Proceedings of the 7th IEEE East-West Design & Test Symposium*, EWDTS '09, pp. 429–432, Moscow, Russian Federation, 18-21 Sept. 2009.

[28] M. Caramia, M. Fabiano, A. Miele, R. Piazza, and P. Prinetto. Automated synthesis of EDACs for FLASH memories with user-selectable correction capability. *Proceedings of IEEE International High Level Design Validation and Test Workshop (HLDVT)*, pp. 113 –120, june 2010.

[29] João MP Cardoso and Michael Hübner. *Reconfigurable Computing: From FPGAs to Hardware/software Codesign.* Springer, 2011.

[30] 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.

[31] R.-A. Cernea, Long Pham, F. Moogat, Siu Chan, Binh Le, Yan Li, Shouchang Tsao, Tai-Yuan Tseng, Khanh Nguyen, J. Li, Jayson Hu, Jong Hak Yuh, C. Hsu, Fanglin Zhang, T. Kamei, H. Nasu, P. Kliza, Khin Htoo, J. Lutze, Yingda Dong, M. Higashitani, Junnhui Yang, Hung-Szu Lin, V. Sakhamuri, A. Li, Feng Pan, S. Yadala, S. Taigor, K. Pradhan, J. Lan, J. Chan, T. Abe, Y. Fukuda, H. Mukai, K. Kawakami, C. Liang, T. Ip, Shu-Fen Chang, J. Lakshmipathi, S. Huynh, D. Pantelakis, M. Mofidi, and K. Quader. A 34 MB/s MLC write throughput 16 Gb nand with all bit line architecture on 56 nm technology. *Solid-State Circuits, IEEE Journal of*, vol. 44, num. 1, pp. 186–194, 2009.

[32] Li-Pin Chang. On efficient wear leveling for large-scale flash-memory storage systems. *SAC 2007: Proceedings of the ACM Symposium on Applied Computing*, pp. 1126–1130, Seoul, Korea, 11–15 Mar. 2007.

[33] Li-Pin Chang and Tei-Wei Kuo. An efficient management scheme for large-scale flash-memory storage systems. *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, pp. 862–868, New York, NY, USA, 2004. ACM.

[34] Yuan-Hao Chang, Jen-Wei Hsieh, and Tei-Wei Kuo. Endurance enhancement of flash-memory storage systems: an efficient static wear leveling design. *DAC 2007: Proceedings of the 44th IEEE Design Automation Conference*, pp. 212–217, San Diego, CA, USA, 4–8 Jun. 2007.

[35] Bainan Chen, Xinmiao Zhang, and Zhongfeng Wang. Error correction for multi-level NAND flash memory using Reed-Solomon codes. *SiPS 2008: Proceedings of the IEEE Workshop on Signal Processing Systems*, pp. 94–99, Washington, DC, USA, 8–10 Oct. 2008.

[36] E. Chen and T. Yen. Comparing SLC and MLC flash technologies and structure. Retrieved February 24, 2013 from the World Wide Web `http://www.advantech.com.tw/epc/newsletter/Whitepaper/WhitePaper_Comparing_SLC_and_MLC_Flash_Technologies_and_Structure_200909.pdf`, Advantech, 2009.

[37] 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*, pp. 53–58, 2009.

[38] T.H. Chen, Y.Y. Hsiao, Y.T. Hsing, and C.W. Wu. An adaptive-rate error correction scheme for NAND flash memory. *VTS 2009: Proceedings of the 27th IEEE VLSI Test Symposium*, pp. 53–58, Santa Cruz, CA, USA, 3–7 May. 2009.

[39] Yanni Chen and Keshab K. Parhi. Small area parallel Chien search architectures for long BCH codes. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12, 545–549, 2004.

[40] Yuan Chen. Flash memory reliability nepp 2008 task final report. Retrieved February 24, 2013 from the World Wide Web `http://trs-new.jpl.nasa.gov/dspace/bitstream/2014/41262/1/09-9.pdf`, 2008.

[41] Mei-Ling Chiang, Paul C. H. Lee, and Ruei-Chuan Chang. Using data clustering to improve cleaning performance for flash memory. *Software: Practice and Experience*, vol. 29, pp. 267–290, 1999.

[42] R. Chien. Cyclic decoding procedures for Bose-Chaudhuri-Hocquenghem codes. *IEEE Transactions on Information Theory*, 10, 4, 357–363, oct 1964.

[43] Sau-Kwo Chiu, Jen-Chieh Yeh, Chih-Tsun Huang, and Cheng-Wen Wu. Diagonal test and diagnostic schemes for flash memories. *Proc. International Test Conference*, pp. 37–46, 7–10 Oct. 2002.

[44] Junho Cho and Wonyong Sung. Strength-reduced parallel chien search architecture for strong BCH codes. *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 55, num. 5, pp. 427–431, 2008.

[45] H. Choi, W. Liu, and W. Sung. VLSI implementation of BCH error correction for multilevel cell NAND flash memory. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 18, 843–847, 2010.

[46] cmp.imag.fr. Cmp project. web available resource - `http://cmp.imag.fr/`, 2012.

[47] J. Cooke. The inconvenient truths of NAND flash memory. *Micron MEMCON*, vol. 7, 2007.

[48] Jim Cooke. The inconvenient truths of NAND flash memory. `http://download.micron.com/pdf/presentations/events/flash_mem_summit_jcooke_inconvenient_truths_nand.pdf`, 2007.

[49] SanDisk Corporation. Sandisk flash-memory cards wear leveling. Technical Report 80-36-00278, October 2003.

[50] M.A. d'Abreu. Nand flash memory – product trends, technology overview, and technical challenges. *ATS 2011: Proceedings of the 20th Asian Test Symposium*, pp. 463–463, New Delhi, India, 20–23 Nov., 2011.

[51] R. Dan and R. Singer. Implementing MLC NAND flash for cost-effective, high-capacity memory. Technical Report 91-SR-014-02-8L, 2003.

[52] Eric Deal. Trends in NAND flash memory error correction. `http://cyclicdesign.com/whitepapers/Cyclic_Design_NAND_ECC.pdf`, 2011.

[53] S. Di Carlo, M. Fabiano, M. Indaco, and P. Prinetto. Design and optimization of adaptable BCH codecs for NAND flash memories. *Elsevier Microprocessors and Microsystems (MICPRO)*, vol. 37, num. 4–5, pp. 407–419, 2013.

[54] S. Di Carlo, M. Fabiano, M. Indaco, and P. Prinetto. ADAGE: an automated synthesis tool for adaptive BCH-based ECC IP-Cores. *ITC 2012: Proceedings of the 43rd IEEE International Test Conference*, p. 15, Anaheim, CA, USA, 4–9 Nov. 2012.

[55] S. Di Carlo, M. Fabiano, P. Prinetto, and M. Cramia. *Design Issues and Challenges of File Systems for Flash Memories*, chapter 1, pp. 3–30. InTech, 2011.

[56] S. Di Carlo, S. Galfano, M. Indaco, and P. Prinetto. Ef3S: An evaluation framework for flash-based systems. *IOLTS 2013: Proceedings of the 19th IEEE International On-Line Testing Symposium*, pp. 199–204, Chania, Creta, 8–10 Jul., 2013.

[57] N. Duann. Error correcting techniques for future NAND flash memory in SSD applications. Retrieved February 24, 2013 from the World Wide Web `http://www.bswd.com/FMS09/FMS09-201-Duann.pdf`, 2009.

[58] M. Fabiano. *Dependability Assessment of NAND Flash-memory for Mission-critical Applications.* PhD thesis, Politecnico di Torino, 2013.

[59] M. Fabiano and G. Furano. Nand flash storage technology for mission-critical space applications. *accepted for publication on IEEE Aerospace and Electronic Systems Magazine (AESS)*, -, 2013.

[60] Michele Fabiano, Marco Indaco, Stefano Di Carlo, and Paolo Prinetto. Design and optimization of adaptable BCH codecs for NAND flash memories. *Microprocessors and Microsystems*, 37, 4–5, 407 – 419, 2013.

[61] R. H. Fowler and L. Nordheim. Electron emission in intense electric fields,. *Proceedings of the Royal Society of London*, volume 119, pp. 173–181, 1928.

[62] M. J. E. Golay. Notes on digital coding. *Proceedings of The IEEE*, 37, 657, 1949.

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

[64] Rw Hamming. Error Detecting and Error Correcting Codes. *Bell System Technical Journal*, 26, 147–160, 1950.

[65] S. Hellmold. The evolving nand flash business model for ssd. *Flash Storage Summits, 2010*, 2010.

[66] J. Henkel, L. Bauer, M. Hübner, and A. Grudnitsky. i-core: A run-time adaptive processor for embedded multi-core systems. *International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, 2011.

[67] Yea-Ling Horng, Jing-Reng Huang, and Tsin-Yuan Chang. A realistic fault model for flash memories. *ATS '00: Proceedings of the 9th Asian Test Symposium*, p. 274, 2000.

[68] IEEE Standards Department. IEEE standard definitions and characterization of floating gate semiconductor arrays. *IEEE Std 1005-1998*, 1998.

[69] D. Ielmini. Reliability issues and modeling of flash and post-flash memory. *Microelectronic Engineering*, 86, 1870–1875, 2009.

[70] D. Ielmini, A.S. Spinelli, A.L. Lacaita, and A. Modelli. A statistical model for silc in flash memories. *IEEE Transactions on Electron Devices*, vol. 49, num. 11, pp. 1955–1961, 2002.

[71] D. Ielmini, A.S. Spinelli, A.L. Lacaita, and M.J. van Duuren. Defect generation statistics in thin gate oxides. *IEEE Transactions on Electron Devices*, vol. 51, num. 8, pp. 1288–1295, 2004.

[72] D. Ielmini, A.S. Spinelli, A.L. Lacaita, and A. Visconti. Statistical profiling of silc spot in flash memories. *IEEE Transactions on Electron Devices*, vol. 49, num. 10, pp. 1723–1728, 2002.

[73] Intel. Understanding the Flash Translation Layer (FTL) specification, AP-684 (order 297816). Retrieved February 24, 2013 from the World Wide Web `http://staff.ustc.edu.cn/~jpq/paper/flash/2006-Intel%20TR-Understanding%20the%20flash%20translation%20layer%20(FTL)%20specification.pdf`, Dec. 1998.

[74] Lee Jae-Duk, Hur Sung-Hoi, and Choi Jung-Dal. Effects of floating-gate interference on NAND flash memory cell operation. *IEEE Electron Device Letters*, 23, 264–266, 2002.

[75] Jae Duk Lee, Sung Hoi Hur, Jung Dal Choi. Effects of floating gate interference on nand flash memory cell operation. *IEEE ELECTRON DEVICE LETTERS*, 23, 5, 264–266, May 2002.

[76] Yeh Jen-Chieh, Wu Chi-Feng, Cheng Kuo-Liang, Chou Yung-Fa, Huang Chih-Tsun, and Wu Cheng-Wen. Flash memory built-in self-test using march-like algorithms. *Proceedings of the First IEEE International Workshop on Electronic Design, Test and Applications*, pp. 137–141, Christchurch , New Zealand, 29-31 Jan. 2002.

[77] Hsieh Jen-Wei, Tsai Yi-Lin, Kuo Tei-Wei, and Lee Tzao-Lin. Configurable flash-memory management: Performance versus overheads. *IEEE Trans. on Computers*, 57, 11, 1571–1583, Nov. 2008.

[78] Sung-Min Joe, Jeong-Hyong Yi, Sung-Kye Park, Hyungcheol Shin, Byung-Gook Park, Young-June Park, and Jong-Ho Lee. Threshold voltage fluctuation by random telegraph noise in floating gate nand flash memory string. *IEEE Transactions on Electron Devices*, vol. 58, num. 1, pp. 67–73, 2011.

[79] Cho Junho and Sung Wonyong. Efficient software-based encoding and decoding of BCH codes. *IEEE Transactions on Computers*, 58, 7, 878–889, July 2009.

[80] Y.H. Kang, J.K. Kim, S.W. Hwang, J.Y. Kwak, J.Y. Park, D. Kim, C.H. Kim, J.Y. Park, Y.T. Jeong, J.N. Baek, et al. High-voltage analog system for a mobile nand flash. *IEEE Journal of Solid-State Circuits*, vol. 43, num. 2, pp. 507–517, 2008.

[81] Jeffrey Katcher. Postmark: a new file system benchmark. Network Appliance Tech Report TR3022, October 1997.

[82] Ki-Tae Park, Myounggon Kang, Soonwook Hwang, Doogon Kim, Hoosung Cho, Youngwook Jeong, Yong-Il Seo, Jaehoon Jang, Han-Soo Kim, Yeong-Taek Lee, Soon-Moon Jung, Changhun Kim. A fully performance compatible 45nm 4-gigabit three dimensional double stacked multi-level nand flash memory with shared bit line structure. *IEEE JOURNAL OF SOLID STATE CIRCUITS*, 44, 1, 208–216, January 2009.

[83] Y. Kumagai, A. Teramoto, T. Inatsuka, R. Kuroda, T. Suwa, S. Sugawa, and T. Ohmi. Evaluation for anomalous stress-induced leakage current of gate SiO_2 films using array test pattern. *IEEE Transactions on Electron Devices*, vol. 58, num. 10, pp. 3307–3313, 2011.

[84] Hye-Jin Lee, Tae-Sung Yoon, Won-Sang Ra, and Jin-Bae Park. Practical pinch detection algorithm for low-cost anti-pinch window control system. *ICIT 2005: Proceedings of IEEE International Conference on Industrial Technology*, pp. 995–1000, Budapest, Hungary, 14–17 Dec. 2005.

[85] Kijun Lee, Sejin Lim, and Jaehong Kim. Low-cost, low-power and high-throughput BCH decoder for NAND Flash Memory. *IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 413–415, may 2012.

[86] Seungjae Lee, Young-Taek Lee, Wook-Kee Han, Dong-Hwan Kim, Moo-Sung Kim, Seung-Hyun Moon, Hyun Chul Cho, Jung-Woo Lee, Dae-Seok Byeon, Young-Ho Lim, Hyung-Suk Kim, Sung-Hoi Hur, and Kang-Deog Suh. A 3.3 v 4 gb four-level nand flash memory with 90 nm cmos technology. *Solid-State Circuits Conference, 2004. Digest of Technical Papers. ISSCC. 2004 IEEE International*, pp. 52–513 Vol.1, 2004.

[87] S. Lin and D. Costello. *Error Control Coding: Fundamentals and Applications*. 2004.

[88] Rich Liu, Hang-Ting Lue, K. C Chen, and Chih-Yuan Lu. Reliability of barrier engineered charge trapping devices for sub-30nm nand flash. *IEDM 2009: Proceedings of the IEEE International Electron Devices Meeting*, pp. 1–4, Baltimore, MD, USA, 7–9 Dec., 2009.

[89] 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*, pp. 303 – 308, oct. 2006.

[90] M-Systems. Flash-memory Translation Layer for NAND flash (NFTL). Retrieved February 24, 2013 from the World Wide Web `http://www.freepatentsonline.com/5404485.pdf`, 1998.

[91] C. Manning. How YAFFS handles NAND errors. Retrieved February 24, 2013 from the World Wide Web http://yaffs.net/gitweb?p=yaffs-docs;a=blob;f=NANDFailureMitigation.odt, 2011.

[92] The Mathworks. Communication System Toolbox. http://www.mathworks.nl/help/comm/ref/primpoly.html, 2012.

[93] S. Medardoni, M. Lajolo, and D. Bertozzi. Variation tolerant noc design by means of self-calibrating links. *Design, Automation and Test in Europe, 2008. DATE'08*, pp. 1402–1407. IEEE, 2008.

[94] MemoryTechnologyDevice. Memory technology device (MTD). Retrieved February 24, 2013 from the World Wide Web http://www.linux-mtd.infradead.org/, 2010.

[95] R. Merritt. Flash will ride dram bus in 2014, says micron. Retrieved September 18, 2013 from the World Wide Web http://www.eetimes.com/document.asp?doc_id=1263046&page_number=2, 2013.

[96] C. Miccoli, J. Barber, C.M. Compagnoni, G.M. Paolucci, J. Kessenich, A.L. Lacaita, A.S. Spinelli, R.J. Koval, and A. Goda. Resolving discrete emission events: A new perspective for detrapping investigation in NAND flash memories. *IRPS 2013: Proceedings of the IEEE International Reliability Physics Symposium*, pp. 3B.1.1–3B.1.6, Anaheim, CA, USA, 14–18 Apr., 2013.

[97] C. Miccoli, C. Monzio Compagnoni, A.S. Spinelli, and A.L. Lacaita. Investigation of the programming accuracy of a double-verify ISPP algorithm for nanoscale NAND flash memories. *IRPS 2011: IEEE International Proceedings of Reliability Physics Symposium*, pp. MY.5.1 – MY.5.6, 10–14 Apr. 2011.

[98] R. Micheloni, L. Crippa, and A. Marelli. *Inside NAND flash memories*. Springer Verlag, 2010.

[99] R. Micheloni, A. Marelli, and K. Eshghi. *Inside Solid State Drives (SSDs)*. Springer Verlag, 2013.

[100] R. Micheloni, A. Marelli, and K. Eshghi. *Inside Solid State Drives (SSDs)*. Springer Series in Advanced Microelectronics, 2013.

[101] R. Micheloni, A. Marelli, and R. Ravasio. *Error correction codes for non-volatile memories.* Springer Verlag, 2008.

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

[103] R. Micheloni, R. Ravasio, A. Marelli, E. Alice, V. Altieri, A. Bovino, L. Crippa, E. Di Martino, L. D'Onofrio, A. Gambardella, E. Grillea, G. Guerra, D. Kim, C. Missiroli, I. Motta, A. Prisco, G. Ragone, M. Romano, M. Sangalli, P. Sauro, M. Scotti, and S. Won. A 4Gb 2b/cell NAND flash memory with embedded 5b BCH ECC for 36MB/s system read throughput. *ISSCC 2006: Proceedings of the IEEE Solid-State Circuits Conference*, pp. 497–506, San Francisco, CA, USA, 6–9 Feb. 2006.

[104] Micron. Hamming codes for NAND flash-memory devices overview. `http://download.micron.com/pdf/technotes/nand/tn2908.pdf`, 2011.

[105] Micron. A trillion bits on a fingertip. Retrieved February 24, 2013 from the World Wide Web `http://www.micron.com/about/blogs/2011/december/a-trillion-bits-on-a-fingertip`, 2011.

[106] N. Mielke, T. Marquart, N. Wu, J. Kessenich, H. Belgal, E. Schares, F. Trivedi, E. Goodness, and L.R. Nevill. Bit error rate in NAND flash memories. *IRPS 2008: Proceedings of the IEEE International Reliability Physics Symposium*, pp. 9–19, Phoenix, AZ, USA, 27 Apr.–1 May. 2008.

[107] N. Mielke, T. Marquart, N. Wu, J. Kessenich, H. Belgal, E. Schares, F. Trivedi, E. Goodness, and L.R. Nevill. Bit error rate in NAND flash memories. *IRPS 2008: Proceedings of the IEEE International Reliability Physics Symposium*, pp. 9–19, Phoenix, AZ, USA, 27 Apr.–1 May. 2008.

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

[109] M.G. Mohammad, K. K. Saluja, and Alex S. Yap. Fault models and test procedures for flash memory disturbances. *J. Electron. Test.*, 17, 6, 495–508, 2001.

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

[111] M.G. Mohammad, K.K. Saluja, and A. Yap. Testing flash memories. *Proceeding of the Thirteenth International Conference on VLSI Design*, pp. 406–411, Calcutta, India, 4-7 Jan. 2000. IEEE Computer Society.

[112] M.G. Mohammad and Laila Terkawi. Fault collapsing for flash memory disturb faults. *ETS '05: Proceedings of the 10th IEEE European Symposium on Test*, pp. 142–147, Washington, DC, USA, 2005. IEEE Computer Society.

[113] V. Mohan, S. Gurumurthi, and M.R. Stan. Flashpower: A detailed power model for nand flash memory. *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2010*, pp. 502–507. IEEE, 2010.

[114] G. Molas, D. Deleruyelle, B. De Salvo, G. Ghibaudo, M. Gely, S. Jacob, D. Lafond, and S. Deleonibus. Impact of few electron phenomena on floating-gate memory reliability. *IEDM 2004: Proceedings of the IEEE International Electron Devices Meeting*, pp. 877–880, 15–15 Dec., 2004.

[115] G. Molas, D. Deleruyelle, B. De Salvo, Gerard Ghibaudo, M. GelyGely, L. Perniola, D. Lafond, and S. Deleonibus. Degradation of floating-gate memory reliability by few electron phenomena. *IEEE Transactions on Electron Devices*, vol. 53, num. 10, pp. 2610–2619, 2006.

[116] M. Momodomi, T. Tanaka, Y. Iwata, Y. Tanaka, H. Oodaira, Y. Itoh, R. Shirota, K. Ohuchi, and F. Masuoka. A 4 mb nand eeprom with tight programmed vt distribution. *IEEE Journal of Solid-State Circuits*, vol. 26, num. 4, pp. 492–496, 1991.

[117] C. Monzio Compagnoni, A.S. Spinelli, S. Beltrami, M. Bonanomi, and A. Visconti. Cycling effect on the random telegraph noise instabilities of nor and nand flash arrays. *IEEE Electron Device Letters*, vol. 29, num. 8, pp. 941–943, 2008.

[118] Todd K. Moon. *Error Correction Coding: Mathematical Methods and Algorithms*. Wiley-Interscience, 2005.

[119] ONFI. Open NAND flash interface (ONFi) specification. Retrieved February 24, 2013 from the World Wide Web `http://www.onfi.org/~/media/ONFI/specs/onfi_3_1_spec.pdf`, 2010.

[120] Yangyang Pan, Guiqiang Dong, and Tong Zhang. Exploiting memory device wear-out dynamics to improve NAND flash memory system performance. *FAST 2011:*

Proceedings of the 9th USENIX conference on File and stroage technologies, pp. 18–18, San Jose, CA, USA, 15–17 Feb. 2011.

[121] G.M. Paolucci, C. Miccoli, C.M. Compagnoni, L. Crespi, A.S. Spinelli, and A.L. Lacaita. Investigation of cycling-induced vt instabilities in nand flash cells via compact modeling. *IMW 2012: Proceedings of the 4th IEEE International Memory Workshop*, pp. 1–4, Milan, Italy, 20–23 May, 2012.

[122] K. Rajesh Shetty, U. Sripati, H. Prashantha Kumar, and B. Shankarananda. Synthesis of BCH codes for enhancing data integrity in flash memories. *International Conference on Industrial and Information Systems (ICIIS)*, pp. 119–124, 29 2010-aug. 1 2010.

[123] I. S. Reed and G. Solomon. Polynomial Codes Over Certain Finite Fields. *Journal of the Society for Industrial and Applied Mathematics*, 8, 300–304, 1960.

[124] Samsung. XSR1.5 bad block management. Retrieved February 24, 2013 from the World Wide Web http://www.findthatpdf.com/download.php?i=4450573&t=hPDF, 2007.

[125] Samsung. XSR1.5 wear leveling. Retrieved February 24, 2013 from the World Wide Web http://en.pudn.com/dl.asp?id=965593, 2007.

[126] Ltd Samsung Electronics Co. NAND flash ECC algorithm (error checking & correction). Retrieved February 24, 2013 from the World Wide Web http://www.elnec.com/sw/samsung_ecc_algorithm_for_256b.pdf, June 2004.

[127] Spansion. What types of ECC should be used on flash-memory? Retrieved February 24, 2013 from the World Wide Web http://www.spansion.com/Support/Application%20Notes/Types_of_ECC_Used_on_Flash_AN.pdf, 2011.

[128] A. Spessot, A. Calderoni, P. Fantini, A.S. Spinelli, C.M. Compagnoni, F. Farina, A.L. Lacaita, and A. Marmiroli. Variability effects on the vt distribution of nanoscale nand flash memories. *Reliability Physics Symposium (IRPS), 2010 IEEE International*, pp. 970–974. IEEE, 2010.

[129] Kang-Deog Suh, Byung-Hoon Suh, Young-Ho Lim, Jin-Ki Kim, Young-Joon Choi, Yong-Nam Koh, Sung-Soo Lee, Suk-Chon Kwon, Byung-Soon Choi, Jin-Sun Yum,

Jung-Hyuk Choi, Jang-Rae Kim, and Hyung-Kyu Lim. A 3.3 v 32 mb nand flash memory with incremental step pulse programming scheme. *IEEE Journal of Solid-State Circuits*, vol. 30, num. 11, pp. 1149–1156, 1995.

[130] H. Sun, B. Wood, and P. Grayson. Qualifying reliability of solid-state storage from multiple aspects. *7th IEEE International Workshop on Storage Network Architecture and Parallel I/O*, 2011.

[131] Sheng-Jie Syu and Jing Chen. An active space recycling mechanism for flash storage systems in real-time application environment. *Proc. 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pp. 53–59, 17–19 Aug. 2005.

[132] Richard Tervo. EE4253 Digital Communications. `http://www.ee.unb.ca/cgi-bin/tervo/bch.pl`, 2010.

[133] Tahui Wang, Nian-Kai Zous, and Chih-Chieh Yeh. Role of positive trapped charge in stress-induced leakage current for flash eeprom devices. *IEEE Transactions on Electron Devices*, vol. 49, num. 11, pp. 1910–1916, 2002.

[134] Wilson. The new and improved filebench. *File and Storage Technologies (FAST), 2008. 6th USENIX Conference on*, 2008.

[135] D. Woodhouse. JFFS : The journalling flash file system. *Proceedings of the Ottawa Linux Symposium*, Ottawa, Ontario Canada, 26-29 July 2001.

[136] Yu Xin, Rong Chun-ming, and Huang Ben-xiong. A flexible garbage collect algorithm for flash storage management. *Proc. Second International Conference on Future Generation Communication and Networking FGCN '08*, volume 1, pp. 354–357, 13–15 Dec. 2008.

[137] E. Yaakobi, J. Ma, A. Caulfield, L. Grupp, S. Swanson, P.H. Siegel, and Wolf J.K. Error correction coding for flash memories. `http://cmrr.ucsd.edu/research/documents/Number31Winter2009_000.pdf`, 2009.

[138] E. Yaakobi, J. Ma, A. Caulfield, L. Grupp, S. Swanson, P.H. Siegel, and J.K. Wolf. Error correction coding for flash memories. *Flash Memory Summit*, 2009.

[139] C. Yang, Y. Emre, C. Chakrabarti, and T. Mudge. Flexible product code-based ECC schemes for MLC NAND flash memories. *Signal Processing Systems (SiPS), 2011 IEEE Workshop on*, pp. 255–260, 2011.

[140] J. C. Yeh, Kuo-Liang Cheng, Yung-Fa Chou, and Cheng-Wen Wu. Flash memory testing and built-in self-diagnosis with march-like test algorithms. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26, 6, 1101–1113, June 2007.

[141] Xu Youzhi. Implementation of Berlekamp-Massey algorithm without inversion. *IEEE Proceedings of Communications, Speech and Vision*, vol. 138, num. 3, pp. 138–140, 1991.

[142] Xu Youzhi. Implementation of Berlekamp-Massey algorithm without inversion. *IEEE Proceedings of Communications, Speech and Vision*, vol. 138, num. 3, pp. 138–140, 1991.

[143] Chen Yuan. Flash memory reliability NEPP 2008 task final report. Retrieved February 24, 2013 from the World Wide Web `http://trs-new.jpl.nasa.gov/dspace/bitstream/2014/41262/1/09-9.pdf`, 2008.

[144] C. Zambelli, M. Indaco, M. Fabiano, S. Di Carlo, P. Prinetto, P. Olivo, and D. Bertozzi. A cross-layer approach for new reliability-performance trade-offs in MLC NAND flash memories. *DATE 2012: Proceedings of the IEEE Design, Automation & Test in Europe*, pp. 881–886, Dresden, Germany, 12–16 Mar., 2012.