Chapter 10
BCH and LDPC Error Correction Codes for NAND Flash Memories

Alessia Marelli and Rino Micheloni

Nowadays NAND Flash memories are part of our lives in many ways. The storage world is a completely new world thanks to NAND. As a matter of fact, it wouldn't be possible to have a smartphone without the use of NAND memories as the storage media. After USB keys and digital cameras, Solid State Drives (SSDs) are now the new disruptive application for Flash. Consumer-class ultra-light and ultra-thin laptops require NAND storage, but it is really in the cloud and in enterprise servers that the use of NAND can be a paradigm shift.

Because NAND devices can't be manufactured without defects, the use of Error Correction Codes (ECCs) has always been a common practice. While BCH (Bose-Chaudhuri-Hocquenghem) is a de facto standard for consumer applications, LDPC (Low-Density-Parity-Check) codes are a typical choice in the enterprise world. This is especially true when looking at planar (2D) ultra-scaled (e.g. 15 nm) NAND. Generally speaking, LDPC offers higher correction capabilities, but BCH remains a good solution when bandwidth requirements are very stringent.

As discussed in previous chapters, 3D NAND is becoming a reality in the market. In terms of noise models, 2D and 3D have some commonalities: they are both very complex and they change during the NAND's lifetime!

We do expect 3D NAND to bring new failure models into the game; all the scientists working on ECCs for non-volatile memories will have to put their best effort for getting as close as possible to the Shannon limit.

To set the stage, in this chapter we cover both BCH and LDPC codes. After a brief introduction, we will see the implementation issues when coupling these codes with a "real" NAND communication channel; practical workarounds will also be discussed.

A. Marelli (✉) · R. Micheloni
Performance Storage BU, Microsemi Corporation, Vimercate, Italy
e-mail: alessia.marelli@ieee.org

R. Micheloni
e-mail: rino.micheloni@ieee.org

R. Micheloni (ed.), *3D Flash Memories*, DOI 10.1007/978-94-017-7512-0_10

10.1 Introduction

During life, multiple sources can corrupt the data stored in NAND cells. The most popular way for data recovery, sometimes used in conjunction with other techniques (e.g. signal processing), is the adoption of an error correction code.

ECCs add redundant terms to the message, such that, on the receiver side, it is possible to detect the errors and to recover the message that was “most probably” transmitted. The set of “encoded” data, i.e. data with the added redundant terms, is usually called *codeword*.

In other words, ECC can decrease the native *Raw Bit Error Rate* (RBER) of NAND. Given a RBER as defined in Eq. (10.1)

$$RBER = \frac{Number\ of\ bit\ errors}{Total\ number\ of\ bits} \tag{10.1}$$

and an ECC able to recover t errors, the codeword error rate, sometimes called *Frame Error Rate* (FER) is computed as in Eq. (10.2)

$$\begin{aligned} FER = 1 - \Bigg[& (1 - RBER)^A + \binom{A}{1} RBER (1 - RBER)^{A-1} \\ & + \cdots + \binom{A}{t} RBER^t (1 - RBER)^{A-t} \Bigg] \end{aligned} \tag{10.2}$$

where A is the codeword size.

Figure 10.1 shows the FER for a 1 kB codeword with an ECC able to correct 1, 10, 50 or 100 errors.

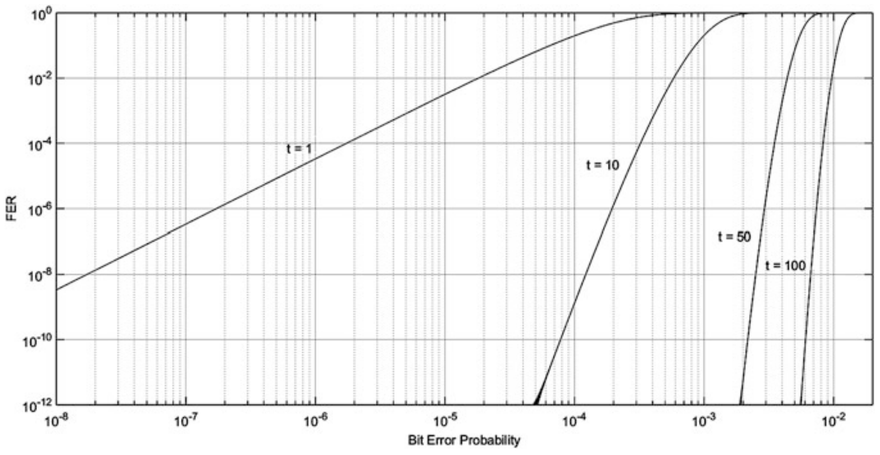

Fig. 10.1 Graph of the frame error probability for a 1 kB codeword with an ECC able to correct 1, 10, 50 and 100 errors

Another quantity used to measure the impact of ECC is the *Uncorrectable Bit Error Rate* (UBER). This is defined as in Eq. (10.3)

$$UBER = \frac{FER}{A} \tag{10.3}$$

A fundamental quantity used to decide which ECC to apply is the Code Rate. The Code Rate is defined as the ratio between the number of protected bits and the total number of transmitted bits (codeword size). If the Code Rate is high, we have few ECC parity bits, i.e. the error correction capability is low. On the other hand, we do not need too much extra space to store them. If the Code Rate is low, we have a higher number of parity bits to protect the data, and the error correction capability is high. In this case, we need more additional space to store the parity bits, and in some cases this is not possible. Even when this is possible, it costs money.

The trade-off between Code Rate and cost ($) is shown in Fig. 10.2. ECC correctability (i.e. the number of correctable bits per codeword) is a function of the Code Rate, as shown in Fig. 10.3. A lower code rate is less efficient in terms of silicon area, but it can recover more errors.

Error correction capability is also influenced by the codeword size (Fig. 10.4). Given the same code rate, the longer the codeword is, the higher the error correction capability is. On the other hand, the longer the codeword is, the more complex the ECC hardware is; a longer latency time to recover the corrupted data is another downside.

In communication theory, *Signal-to-Noise Ratio* (SNR) is usually adopted instead of RBER. Signal-to-noise ratio is a measure that compares the level of a desired signal to the level of the background noise. It is defined as the ratio of the

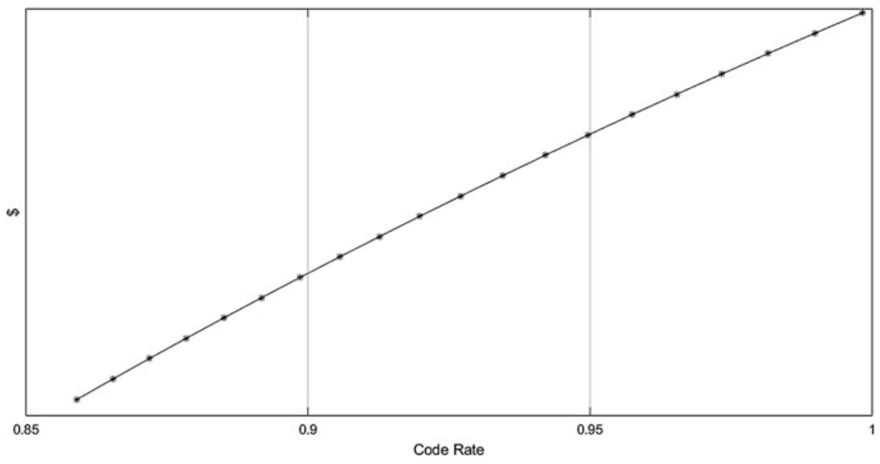

Fig. 10.2 Trade-off between code rate and cost

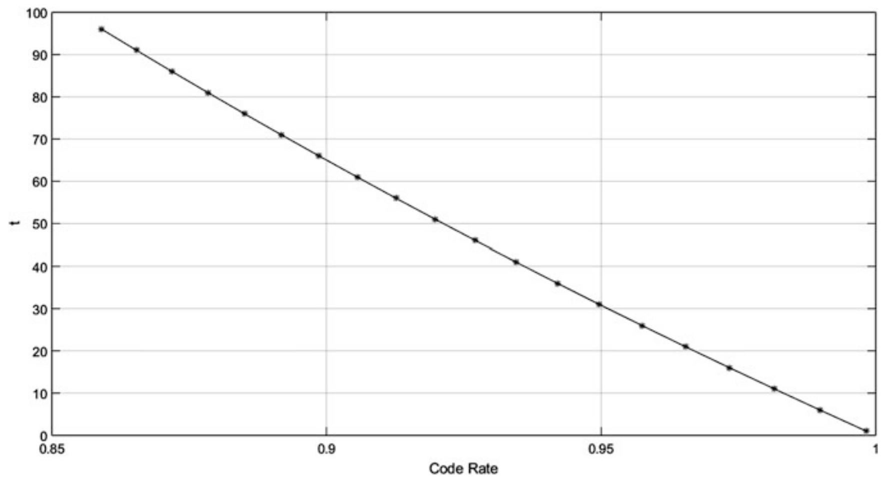

Fig. 10.3 Trade-off between code rate and correctability *t* for a codeword size of 1024 bytes

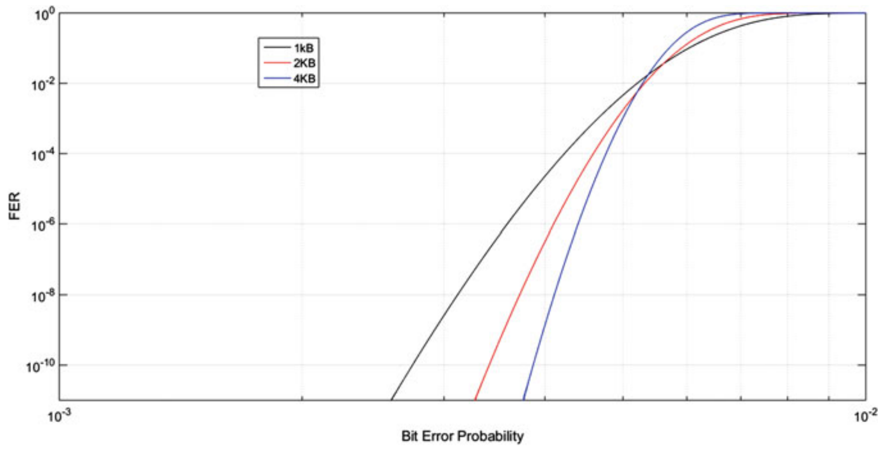

Fig. 10.4 FER versus BER for an ECC with code rate of 0.9 and different codeword sizes

signal power to the noise power, and it is often expressed in decibels. A ratio higher than 1 (i.e. greater than 0 dB) indicates more signal than noise.

Error correction codes belong to the information theory, whose "father" is Shannon; he demonstrated a fundamental theorem known as Shannon Limit [1]. This theorem establishes a limit in terms of achievable signal to noise ratio (SNR) for an error-free communication in a coded system with Code Rate R. The power of Shannon limit is the following: if we can guarantee that SNR does not exceed this limit, then we are sure of the existence of a coded system (with rate R) able to achieve error-free communication. Unfortunately, there isn't any

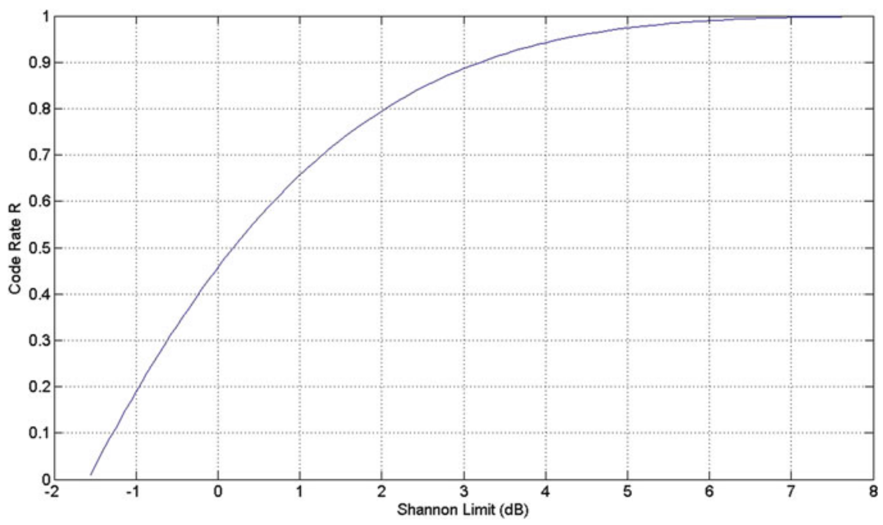

Fig. 10.5 Shannon limit for different code rates

constructive way for building such a coded system; this is why there is an intense research activity for finding codes that get as close as possible to the Shannon limit. The limit can be computed assuming an AWGN (Additive White Gaussian Noise) channel and a BPSK (Binary Phase Shift Keying) modulation (Fig. 10.5).

The achievable SNR can be translated in achievable BER. The Shannon limit is used to evaluate different coded systems: the best code is the one closest to the limit [2, 3].

ECCs can be split in hard decision codes and soft decision codes. This distinction is not based on the structure of the code itself, but on the way the information is treated by the code. A binary hard decision code treats all the data in a digital way, i.e. "0" or "1"; in other words, the analog information is converted into digital format by using one fixed reference level. On the contrary, a soft decision code uses reliability information to take decisions: for example, a "0" is read with a 90 % reliability and a "1" is read with a 10 % reliability. In the following sections, we will see how the soft information applies to NAND Flash, and a comparison between hard and soft codes [3].

Basically, a *Code C* is the set of codewords obtained by associating the q^k messages of length k of the space A to q^k words of length n of the space B in a univocal way. A code is defined as *linear* if, given two codewords, their sum is a codeword. When a code is linear, encoding and decoding can be described with matrix operations.

We define the *generator matrix of a code C* as G. It follows that all the codewords can be obtained as a combination of the rows of G. Therefore, encoding a data message m is equivalent to multiply the message m by the code generator matrix G, according to Eq. (10.4).

$$c = m \cdot G \tag{10.4}$$

G is called in *standard form* or in *systematic form* if $G = (I_k, P)$, where I_k is the identity matrix $k \times k$, and P is a matrix $k \times (n - k)$. If G is in standard form, then the first k symbols of a codeword are called information symbols.

From the matrix G in systematic form, it is straightforward to derive the *parity matrix* $H = (-P^T, \quad I_{n-k})$ where P^T is the transpose of P and it is a matrix $(n - k) \times k$, and I_{n-k} is the identity matrix $(n - k) \times (n - k)$ [4, 5].

Systematic codes have the advantage that the data message can clearly be identified in the codeword and, therefore, it can be read before decoding. For codes in non-systematic form the message is no more recognizable in the encoded sequence and it is necessary to have the inverse encoding function to recognize the data sequence.

If C is a linear code with parity matrix H, then $x \cdot H^T$ is called *syndrome* of x. It follows that all the codewords have a syndrome equal to 0.

The syndrome is the key player of decoding. Once a message r is received (i.e. read from the memory), it is necessary to understand if it has been corrupted by calculating:

$$s = x \cdot H^T \tag{10.5}$$

There are two possibilities:

- $s = 0 \Longrightarrow$ the message r is recognized as correct;
- $s \neq 0 \Longrightarrow$ the received message contains some errors.

In the latter case a decoding procedure starts.

In order to understand how many errors a code is able to correct and detect we need a metric. In the coding theory, it is called *minimum distance* or *Hamming distance* d of a code, and it corresponds to the minimum number of different symbols between any two codewords.

A code has *detection capability* v if it is able to recognize all the messages, containing v errors at the most, as corrupted.

The detection capability is related to the minimum distance as described in Eq. (10.6).

$$v = d - 1 \tag{10.6}$$

A code has *correction capability* t if it is able to correct each combination of a number of errors equal to t at the most. The correction capability is calculated from the minimum distance d with Eq. (10.7):

$$t = \left[\frac{d - 1}{2}\right] \tag{10.7}$$

where the square brackets mean the floor function.

Codes can be manipulated or combined depending on applications. The possible operation to increase the minimum distance of a code is the extension: a code $C[n, k]$ is *extended* to a code $C'[n + 1, k]$ by adding one more parity symbol. Generally speaking, for binary codes, the additional parity bit is the total parity of the message. This is calculated as *sum modulo 2* (XOR) of all the bits of the message.

When the “natural” length of the code does not fit the application constraints (e.g. the NAND Flash page), it is possible to change it with the shortening operation: a $C[n, k]$ is *shortened* into a code $C'[n - j, k - j]$ by erasing j columns of the parity matrix. Please note that the columns deleted are the ones corresponding to the user data. With this operation, the Code Rate is decreased.

A similar operation, but with a very different outcome, is the puncturing operation. Puncturing is the process of removing some of the parity bits after encoding. This has the same effect of encoding with an error correction code with a higher rate. The good things is that the same decoder can be used regardless of how many bits have been punctured; therefore, puncturing considerably increases the flexibility of the system without significantly increasing its complexity [6].

10.2 BCH Codes

BCH codes belong to the most important class of cyclic algebraic codes. They were found through independent researches by Hocquenghem in 1959 and by Bose and Ray-Chauduri in 1960 [7, 8].

For BCH codes the minimum distance can be ensured during construction. The definition of the code itself is based on the distance concept and on the Galois field [9, 10].

Let β be an element of the Galois Field $GF(q^m)$. Let b be a non-negative integer. A BCH code with “designed” distance d is generated by the polynomial $g(x)$ of minimal degree that has $d - 1$ consecutive powers of β: $\beta^b, \beta^{b+1}, \ldots, \beta^{b+d-2}$ as roots. Given ψ_i the minimal polynomial of β^{b+i} for $0 \le i < d - 1$, $g(x)$ is computed as:

$$g(x) = LCM\{\psi_0(x), \psi_1(x), \ldots, \psi_{d-2}(x)\} \tag{10.8}$$

and the protected data size is $k = n - deg(g(x))$.

It is possible to show that the designed d is at least $2t + 1$, hence the code is able to correct t errors.

If we assume $b = 1$, and β a primitive element of $GF(q^m)$, then the code becomes a *narrow-sense* and *primitive* BCH code of length $q^m - 1$ able to correct t errors. We shall now consider narrow-sense primitive BCH codes.

In general, decoding of a BCH code is at least 10 times more complicated than encoding. In this chapter we deal with binary BCH codes only, whose structure is presented in Fig. 10.6.

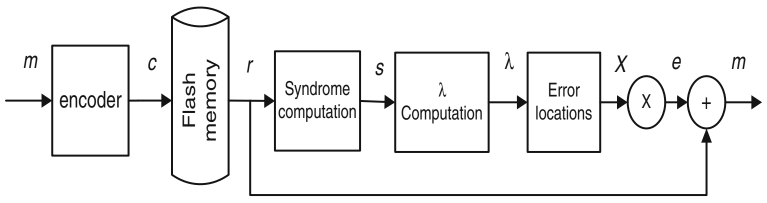

Fig. 10.6 General structure of a binary BCH code

10.2.1 BCH Encoding

Let's assume a BCH code $[n, k]$ with generator polynomial $g(x)$ and a message $m(x)$ to be encoded, which is written as a polynomial of degree $k - 1$.

First of all, the message $m(x)$ is multiplied by x^{n-k} and subsequently divided by $g(x)$, thus obtaining a quotient $q(x)$ and a remainder $r(x)$ in accordance with Eqs. (10.9) and (10.10).

$$\frac{m(x) \cdot x^{n-k}}{g(x)} = q(x) + \frac{r(x)}{g(x)} \tag{10.9}$$

$$m(x) \cdot x^{n-k} + r(x) = q(x) \cdot g(x) \tag{10.10}$$

The multiplication of the message $m(x)$ by x^{n-k} produces, as a result, a polynomial of degree $n - 1$ where the first $n - k$ coefficients, now null, will then be occupied by parity bits.

Therefore, the encoded word $c(x)$ is calculated as:

$$c(x) = m(x) \cdot x^{n-k} + r(x) \tag{10.11}$$

Practical implementation of Eq. (10.11) is depicted in Fig. 10.7. Please note that, since we are considering binary BCH codes, the sum is actually a XOR, while the product is an AND.

The "natural" structure of BCH encoding is sequential; this is not great in high-speed implementations, because it slowly proceeds by byte, word or double word. Figure 10.7b shows the unrolled implementation, assuming a processing of 1 byte at a time [4]. In the figure it is possible to see that the content of each register does not depend on a single input anymore but on a whole byte.

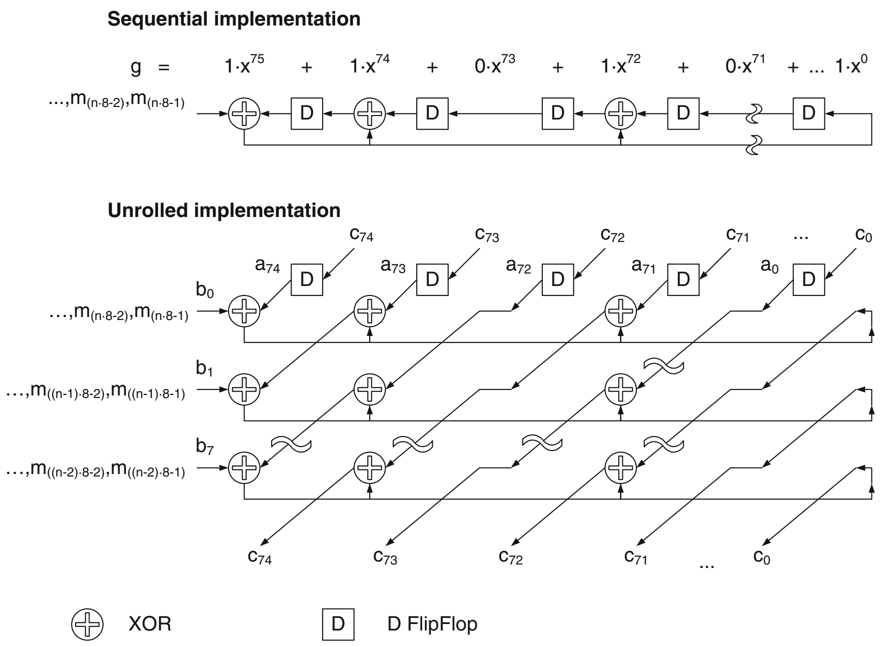

Fig. 10.7 Description of the sequential implementation of the BCH divider: it can be unrolled for a parallel implementation [4]

10.2.2 BCH Decoding

The decoding operation follows three fundamental steps, as shown in Fig. 10.6:

- calculation of the syndromes;
- calculation of the coefficients of the error locator polynomial (usually done with Berlekamp-Massey algorithm [4, 5]);
- calculation of the roots of the error locator polynomial (usually done with Chien algorithm [4, 11]).

During transmission (reading) of the encoded message some errors may occur. Errors can be represented by a polynomial that has coefficient "1" at every error's position:

$$E(x) = E_0 + E_1 x + \cdots + E_{n-1} x^{n-1} \tag{10.12}$$

If ECC can correct t errors, then t non-null coefficients are allowed in Eq. (10.12) at most.

Therefore, the transmitted (read) vector $R(x)$ is:

$$R(x) = c(x) + E(x) \tag{10.13}$$

The first decoding step consists in calculating the $2t$ syndromes for the read message:

$$\frac{R(x)}{\psi_i(x)} = Q_i(x) + \frac{S_i(x)}{\psi_i(x)} \quad \text{with} \quad 1 \le i \le 2t \tag{10.14}$$

$$S_i(x) = Q_i(x) \cdot \psi_i(x) + R(x) \quad \text{with} \quad 1 \le i \le 2t \tag{10.15}$$

In accordance with Eqs. (10.14) and (10.15), the received vector is divided by each minimal polynomial ψ_i forming the generator polynomial, thus getting a quotient $Q_i(x)$ and a remainder $S_i(x)$ called *syndrome*.

At this point the $2t$ syndromes must be evaluated into the elements β, β^2, β^3,..., β^{2t}, whose ψ_i are the minimal polynomials. According to Eq (10.16), this evaluation is the evaluation of the message received in β, β^2, β^3,..., β^{2t}, since $\psi_i(\beta^i) = 0$ (for $1 \le i \le 2t$) because of the definition of minimal polynomial.

$$S_i(\beta^i) = S_i = Q_i(\beta^i) \cdot \psi_i(\beta^i) + R(\beta^i) = R(\beta^i) \tag{10.16}$$

Consequently, the ith syndrome can be calculated either as a remainder of the division between the received message and the minimal polynomial ψ_i, then evaluated in β^i, or as the evaluation in β^i of the received message.

In case there aren't any errors, the received polynomial is a codeword: therefore, the remainder of the division of Eq. (10.14) is null and all the syndromes are identically null. Verifying if the syndromes are identically null is a necessary and sufficient condition to understand if the read message is a codeword (or if some errors occurred).

For binary codes we use the following property:

$$S_{2i} = S_i^2 \tag{10.17}$$

such that we can calculate only t syndromes.

Since the syndromes are computed as the remainder of the division between two polynomials in the Galois field, it is straightforward to understand that the implementation is similar to the one of the encoder.

The Error Locator Polynomial $\Lambda(x)$ is defined as the polynomial whose roots are the inverse of the error positions.

$$\Lambda(x) = \prod_{i=1}^{v} (1 - xX_i) \tag{10.18}$$

The degree of the error locator polynomial gives the number of errors that occurred. As the degree of $\Lambda(x)$ is t at most, in case of more than t errors, $\Lambda(x)$ could erroneously indicate t or less errors.

Coefficients of the error locator polynomial are linked to the syndromes by the following equations

$$\begin{pmatrix} S_{v+1} \\ S_{v+2} \\ S_{v+3} \\ \vdots \\ S_{2v-1} \end{pmatrix} = \begin{pmatrix} S_1 & S_2 & S_3 & \cdots & S_v \\ S_2 & S_3 & S_4 & \cdots & S_{v+1} \\ S_3 & S_4 & \cdots & \cdots & S_{v+2} \\ \vdots & \vdots & \vdots & & \vdots \\ S_v & S_{v+1} & S_{v+2} & \cdots & S_{2v-1} \end{pmatrix} \cdot \begin{pmatrix} \Lambda_v \\ \Lambda_{v-1} \\ \Lambda_{v-2} \\ \vdots \\ \Lambda_1 \end{pmatrix} \tag{10.19}$$

Generally speaking, the method to compute the coefficients of the error locator polynomial is the Berlekamp-Massey algorithm [4, 12].

The philosophy of the Berlekamp algorithm consists in solving the set of Eq. (10.19) in an iterative way by consecutive approximations.

After $2t$ iterations, $\Lambda(x)$ is the error locator polynomial; in the binary case it is possible to perform the Berlekamp algorithm in t iterations. In the following we describe the flow diagram for the inversion-less binary Berlekamp Massey algorithm (Fig. 10.8) [13].

First of all, we define the syndrome polynomial as

$$1 + S = 1 + S_1 z + S_2 z^2 + \cdots + S_{2t-1} z^{2t-1} \tag{10.20}$$

The initial conditions are given as follows:

$$v^{(0)} = 1 \quad k^{(0)} = 1 \quad \text{and} \quad \delta^{(2i)} = 1 \quad \text{if} \quad i < 0 \tag{10.21}$$

We define $d^{(2i)}$ as the coefficient of z^{2i+1} in the product $(1 + S(z))v^{(2i)}(z)$.

- If S_{2i+1} is unknown the algorithm is finished;
- otherwise

$$v^{(2i+2)}(z) = \delta^{(2i-2)} v^{(2i)}(z) + d^{(2i)} k^{(2i)}(z) \cdot z \tag{10.22}$$

$$k^{(2i+2)}(z) = \begin{cases} z^2 k^{(2i)}(\mathrm{z}) & \text{if} \quad d^{(2i)} = 0 \quad \text{or} \quad \text{if} \quad \deg v^{(2i)}(z) > i \\ z v^{(2i)}(\mathrm{z}) & \text{if} \quad d^{(2i)} \neq 0 \quad \text{or} \quad \text{if} \quad \deg v^{(2i)}(z) \leq i \end{cases} \tag{10.23}$$

$$\delta^{(2i)} = \begin{cases} \delta^{(2\mathrm{i}-2)} & \text{if} \quad d^{(2i)} = 0 \quad \text{or} \quad \text{if} \quad \deg v^{(2i)}(z) > i \\ d^{(2i)} & \text{if} \quad d^{(2i)} \neq 0 \quad \text{or} \quad \text{if} \quad \deg v^{(2i)}(z) \leq i \end{cases} \tag{10.24}$$

The roots of $v^{(2t)}(z)$ coincide with those of $\Lambda^{(2t)}(z)$.

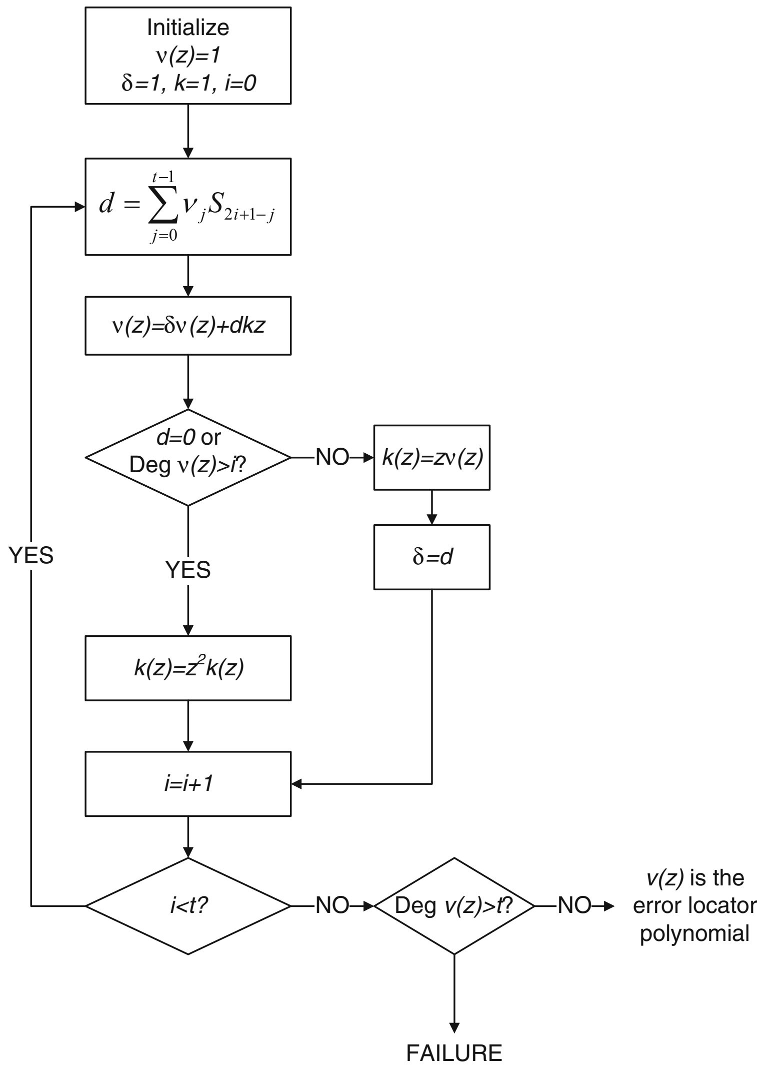

Fig. 10.8 Flow diagram for the inversion-less Berlekamp algorithm

Even if the algorithm is very complex, it usually does not require a parallel implementation, since the size of the memory buffer and the execution latency are acceptable in most of the cases.

The last step of the decoding process consists in searching for the roots of the error locator polynomial, as per Eq. (10.25). If the roots are not coincident and they belong to the Galois field, then it is enough to calculate their inverse to have the error positions. If they are coincident, or they do not belong to the correct field, it

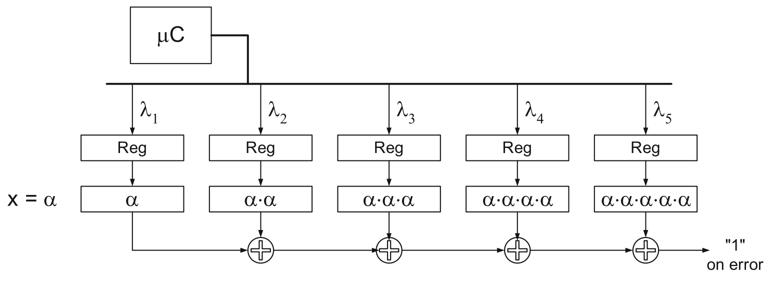

Fig. 10.9 Chien machine for a 5-error BCH: sequential implementation [4]

means that the received message has a distance from a codeword greater than t. In this case an uncorrectable error pattern occurred and the decoding process fails.

$$\Lambda(x) = 1 + \Lambda_1 x + \cdots + \Lambda_t x^t \tag{10.25}$$

To determine the roots of the polynomial, the Chien machine neatly evaluates $\Lambda(x)$ in all the elements of the field α^0, α^1, α^2, α^3,... α^N. For each element i of the field such that the polynomial is null, the corresponding position $(2^m - 1 - i)$ is an error position. A possible implementation of the Chien machine is represented in Fig. 10.9.

10.2.3 Multi-channel BCH

When BCH is used in a NAND-based system such as a Solid State Drive, it is necessary to find out a balance between area and bandwidth. In fact, SSDs run several NAND devices in parallel in order to achieve their target performances of bandwidth and IOPS. Usually, NANDs are split in groups called “Flash Channels”: channels work in parallel and read/write/erase operations can be interleaved within the same channel (Fig. 10.10). In this multi-channel scenario, multiple encoding and decoding machines are necessary, considering that, especially with ultra-scaled geometries and multi-level storage (Chap. 3), correction is required all the time (because of the high RBER).

In order to keep up with the bandwidth requirements, the most straightforward solution would be one encoder and one decoder per channel. However, this approach is extremely area consuming, especially because of the decoder.

As far as the encoding is concerned, it is very important that the data coming from the host (CPU or Operating System) are dispatched to the various channels without latency. There are three possible approaches, starting from the less area consuming:

- single encoder shared among all Flash channels [14];
- a pool of encoders;
- one encoder per channel.

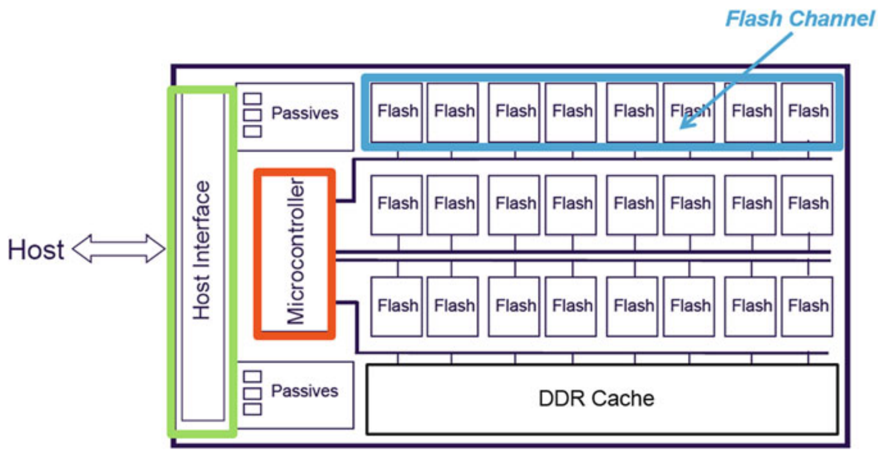

Fig. 10.10 Flash channel inside a solid state drive

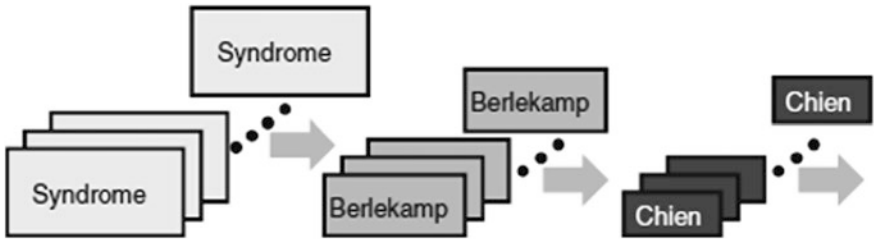

Fig. 10.11 ECC decoding structure for handling multiple channels

As mentioned, the right hardware choice comes from the tradeoff between silicon area and latency.

Let's now move to the decoding phase. The overall structure is shown in Fig. 10.11. In this scenario, number of hardware machines to execute syndrome computation, Berlekamp-Massey algorithm and Chien computation can be different.

Syndrome computation can be treated in the same way as the encoder, since all the read messages require this computation. Execution of the Berlekamp–Massey algorithm is pretty fast because it requires t iterations only.

As described in the previous section, the Chien machine searches for the roots, one at a time. Such operation, carried out for all the bits of the message, results to be very time consuming. Solution is, of course, a parallel architecture. Unlike the parity and the syndromes computation machines, which have to operate with a parallelism equal to the input data parallelism, the Chien machine does not have particular limits other than complexity, area and power consumption. In this parallel

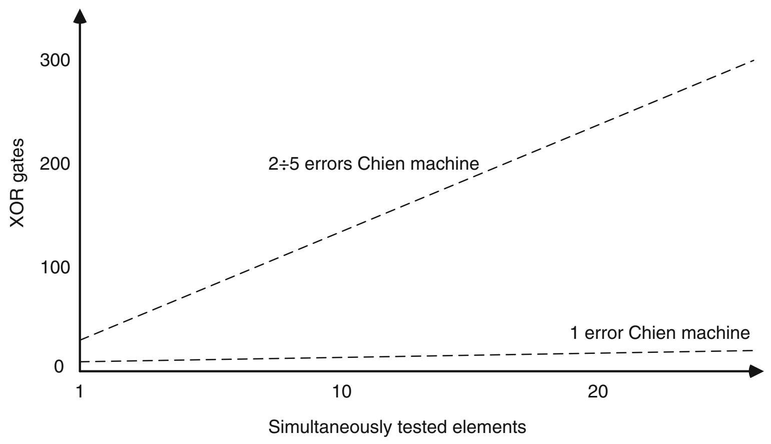

Fig. 10.12 Area impact of the Chien parallelism

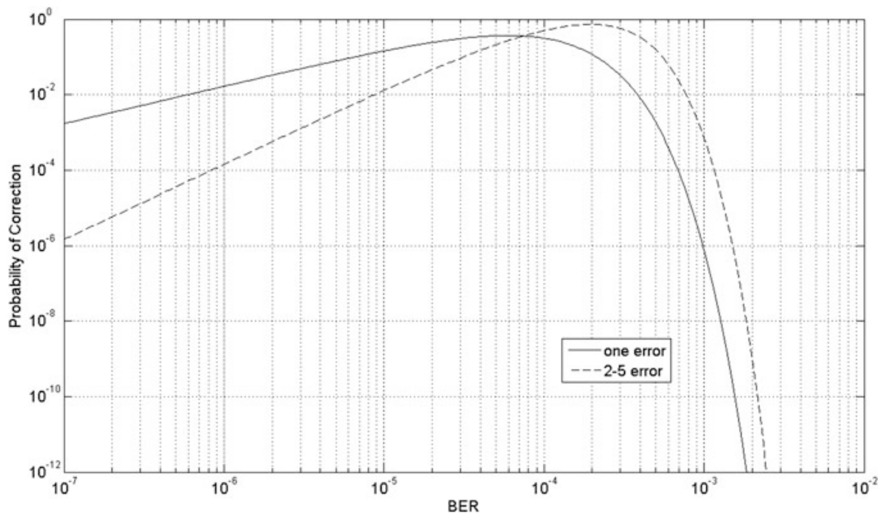

Fig. 10.13 Probability of correction for a 2112-byte page: single error versus 2 ÷ 5 errors

implementation, more error positions are contemporarily evaluated at each computation cycle.

The execution time of the Chien algorithm is usually seen by the system as an additional latency time. If the probability to have one or more errors becomes considerable, this latency can significantly impact the system performance. The downside of the Chien parallelism is the impact on silicon area, as sketched in Fig. 10.12.

Figure 10.13 shows, for a 2112-Byte page, the probability of correcting only one error and the probability of correcting 2 ÷ 5 errors. Assuming a BER of 10^{-6}, we have that the probability of a single error is equal to 1.7×10^{-2} and the probability of 2 ÷ 5 errors is equal to 1.5×10^{-4}, respectively. The probability of a single error is definitely more significant and since the Berlekamp algorithm exactly indicates the number of errors to correct, it may be useful to exploit this information.

The resulting system is, therefore, composed of a couple of Chien machines with different parallelisms, one for the correction of the single error and the second for the correction of 2 ÷ 5 errors (Fig. 10.14).

This solution can be multiplied by any number of machines, especially if the error correction capability t of the BCH we are dealing with is high. In this case, we can compute the frequency of errors t' that is more likely to occur, given the estimated raw bit error rate, and have multiple Chien machine searching for t' roots with a high parallelism. On the contrary, the number of hardware machines to locate the roots of $t^* > t'$ errors can be smaller, and with a smaller parallelism [3, 4].

Of course, the numbers mentioned above are just an example; they might significantly change depending on the NAND technology node and on the number of bits stored within the same physical cell (e.g. MLC or TLC).

10.2.4 Multi-code Rate BCH

As discussed in the introduction of this chapter, it is typical for NAND to deal with noise sources that vary during its lifetime. When NAND is fresh (i.e. few Program/Erase cycles, Chap. 2) and there is no retention, RBER can be pretty low; the situation is totally different at the end of life, i.e. when the device has been read/erased/written multiple times. It follows that it is desirable to have an ECC able to change its correction capability during life.

There are codes for which it is easy to change the code rate, while in other cases it is not that straightforward: BCH is one of them because of its construction. In this section, we present a way for building a multi-code rate BCH with a minimum area overhead.

Encoder is the main issue. As discussed above, parity bits are computed as the remainder of the division between the user data and the generator polynomial, where the latter one is computed as the multiplication among the minimum polynomial of t elements. If we want to adapt a BCH code able to correct t error to correct t' errors, where $t' < t$, the easiest way is to have a second encoder which computes the remainder of the division between the user data and the generator polynomial, where the latter one is computed as the multiplication among the minimum polynomial of t' elements. This approach has a big area overhead, because the encoding area is doubled.

A smarter and a less area consuming way is to derive the parity bits of the t'-code from the parity bits of the t-code. Indeed, for the generator polynomials the equality Eq. (10.26) holds true.

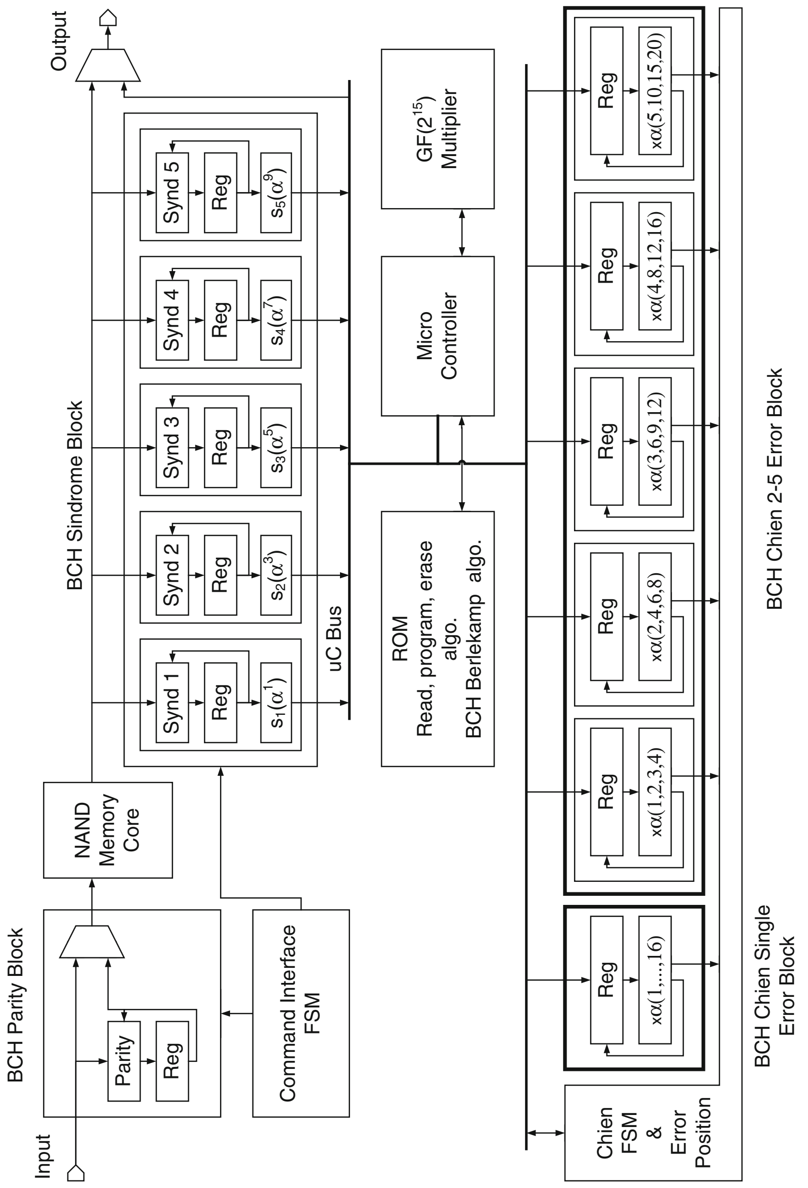

Fig. 10.14 Example of BCH engine including 2 parallel Chien hardware machines [4]

$$g(t, x) = g(t', x) * h(x) \tag{10.26}$$

Parity bits $r(x)$ are computed as remainder of the division between user data $c(x)$ and generator polynomial $g(t, x)$. From the definition of remainder we can write

$$c(x) = q(x) * g(t, x) + r(x) \tag{10.27}$$

where $q(x)$ is the quotient of the division and $deg(r(x)) < deg(g(t, x))$.

The division of $r(x)$ by $g(t', x)$ leads to

$$r(x) = q_1(x) * g(t', x) + r'(x) \tag{10.28}$$

where $q_1(x)$ is the quotient of the division and $deg(r'(x)) < deg(g(t', x))$. By substituting Eqs. (10.26) and (10.28) in Eq. (10.27) we obtain

$$\begin{aligned} c(x) &= q(x) * g(t', x) * h(x) + q_1(x) * g(t', x) + r'(x) \\ &= [q(x) * h(x) + q_1(x)] * g(t', x) + r'(x) \end{aligned} \tag{10.29}$$

It is clear that $r'(x)$ is the remainder of the division between $c(x)$ and $g(t', x)$. The circuit for a multi-code rate BCH encoder is shown in Fig. 10.15.

The overhead for this implementation is a programmable LSFR, which divides the remainder of the first division by a factor of $g(x)$. Of course, we can have more than 2 encoders and multiple programmable LSFRs. Thanks to the LSFR programmability, when NAND is fresh, we can select a BCH code with a small error correction capability, and user data are encoded with two subsequent divisions. When the NAND gets older, we can execute a single division, as the subsequent division is not required anymore.

Decoding is much easier. The syndromes are computed as different divisions by all the factors of $g(t, x)$. If we want to compute the syndromes by using the factors of $g(t', x)$, where $g(t', x)$ is a factor of $g(t, x)$, it is enough to disable the circuits that compute the last $t - t'$ syndromes.

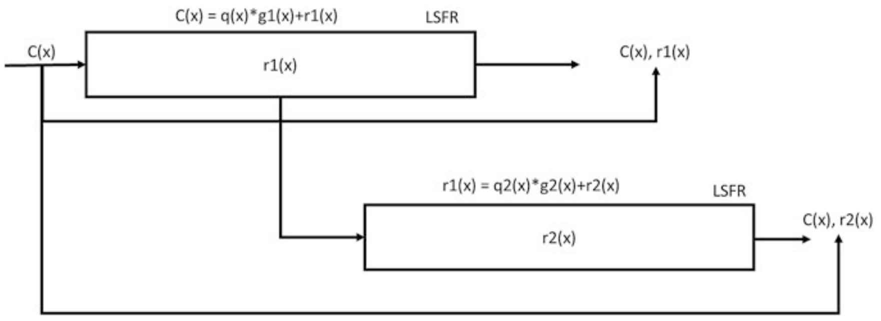

Fig. 10.15 Example of multi-code rate BCH encoder with two different generator polynomials

Berlekamp-Massey algorithm is not impacted by the multi-code rate: it completes in fewer iterations at the beginning of life, with coefficients t' instead of t.

Chien algorithm is not impacted at all. Again, it will stop after finding t' roots instead of t. However, in order to keep up with the SSD's bandwidth, a multi-Chien machine approach (Sect. 10.2.3) is likely to be implemented in a multi-code rate environment.

10.2.5 BCH Detection Properties

BCH codes are not perfect codes: for this reason it is difficult that a codeword with more than t errors moves in the correction sphere of another codeword. The codewords of BCH codes are well separated, and only a number of errors much bigger than t could partially overlap their correction spheres [3]. It follows that the erroneous corrections are made only when the received message is located in a correction sphere different from the original codeword.

Given a binary linear code C able to correct t errors, the probability of miscorrection P_{ME} is defined as the probability that an ideal bounded distance decoder executes erroneous corrections. The weighted probability $P_E(w)$ is the probability of executing erroneous corrections when w errors occur.

Please note that the probability P_{ME} depends on the code C and on the transmission channel.

Theorem 10.2.1 *The weighted probability $P_E(w)$ is computed as:*

$$P_E(w) = \frac{D_w}{\binom{n}{w}} \tag{10.30}$$

where D_w is the number of decodable words and w is in the range $[t + 1, n]$.

The number of decodable words can be computed as

$$D_w = \sum_{i=0}^{n} a_i \sum_{s=0}^{t} N(i, w; s) \tag{10.31}$$

where $N(i, w; s)$ is the number of words with weight w and distance s from a word of weight i. This is computed by Eq. (10.32)

$$N(i, w; s) = \begin{cases} \binom{n-i}{\frac{s+w-i}{2}} \binom{i}{\frac{s-w+1}{2}} & \text{if} \quad |w-i| \leq s \\ 0 & \text{if} \quad |w-i| > s \end{cases} \tag{10.32}$$

By substituting Eq. (10.31) in Eq. (10.30) we have:

$$P_E(w) = \frac{\sum_{i=0}^{n} a_i \sum_{s=0}^{t} N(i,w;s)}{\binom{n}{w}} \tag{10.33}$$

P_{ME} is computed based on $P_E(w)$ as described in Eq. (10.34)

$$P_{ME} = \sum_{w=t+1}^{n} P_E(w)\phi(w) \tag{10.34}$$

where $\phi(w)$ is the probability that a word has weight w.

For a Binary Symmetric Channel BSC

$$P_{ME} = \sum_{w=t+1}^{n} D_w p^w (1-p)^{n-w} \tag{10.35}$$

where p is the bit error probability.

D_w can be computed according to Eq. (10.31). Unfortunately, the weights a_i are unknown for BCH codes and must be estimated.

There are a number of different theorems that can help estimating these weights for a BCH code.

Theorem 10.2.2 Peterson Estimation *The weight $a_i$of a primitive BCH code of length n and error correction capability t can be approximated as*

$$a_i \cong \frac{\binom{n}{i}}{(n+1)^t} \tag{10.36}$$

In order to have upper bounds, different correction terms are added to Eq. (10.36).

Figures 10.16 and 10.17 shows P_E and P_{ME} for BCH[16383, 15851, 77], based on the Peterson estimation. Both P_E and P_{ME} exhibit a monotonic behavior.

It follows that the real P_E and P_{ME} behaviour should be increasingly monotonic with a long floor in the middle [11].

When both the code length and the Code Rate are high, this floor can be approximated with

$$Q = 2^{-(n-k)} \sum_{s=0}^{t} \binom{n}{s} \tag{10.37}$$

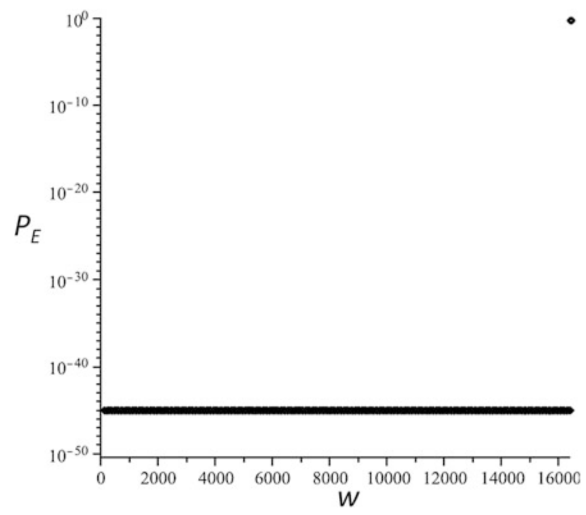

Fig. 10.16 P_E behavior for BCH[16383, 15851, 77] based on the Peterson estimation

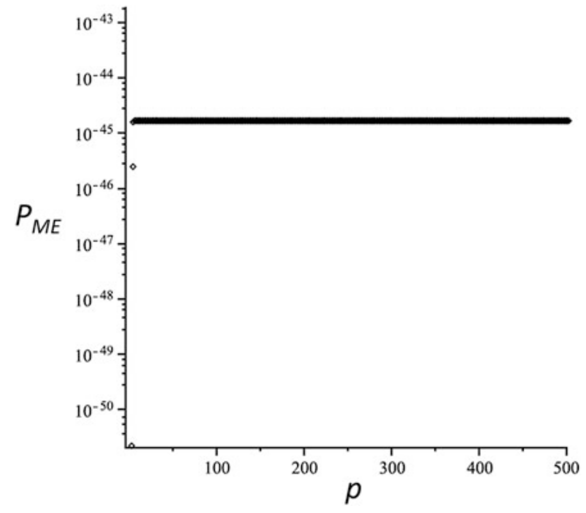

Fig. 10.17 P_{ME} for BCH [16383, 15851, 77] based on the Peterson estimation

To sum up, we can state that the BCH code has a very good detection properties for a long codeword; this feature is well suited for NAND-based systems such as SSDs. In fact, when a catastrophic error occurs or when the error correction capability of the code is overcome, in the vast majority of the cases, BCH signals a decoding failure without attempting erroneous corrections.

Of course, this behaviour becomes really key when BCH is concatenated with another code.

10.3 Low-Density Parity-Check (LDPC) Codes

Since its re-discovery in late 1990s, LDPC code has received a tremendous amount of attentions because of the excellent error-correction capability and experienced a widespread use in many real-life data communication and storage applications. In 1960s Dr. Gallager invented LDPC codes [15], in which two innovative ideas were exploited: iterative decoding and constrained random code construction.

LDPC codes are known as "capacity approaching codes"; in other words, they are a category of codes able to reach a Frame Error Rate very close to the Shannon limit. The main reason is the powerful soft decoding, as shown in Figs. 10.18 and 10.19. Figure 10.18 shows the Shannon limit for 2 BCH codes and 2 hard decoded LDPC codes. In this case, LDPC doesn't show any significant advantage, mainly because of two reasons: the use of hard instead of soft, and the adopted decoding algorithm (i.e. bit-flipping) [5]. LDPC is the clear winner in Fig. 10.19, thanks to the soft decoding. To be fair, the truth is that soft decoding pushes away the Shannon limit; a careful review of the graph reveals that soft LDPC is very close to the Hard Shannon limit, but still far from the Soft Shannon limit.

LDPC are block linear codes defined with a very sparse parity check matrix H. Each matrix can be translated into its corresponding Tanner graph, where there is a number of parity checks equal to the number of the matrix rows called "check nodes"; there is also a number of variable nodes equal to the number of matrix columns. A check node is connected to a variable node if there is a "1" in the corresponding position in the matrix H.

$$H = \begin{pmatrix} 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & \textcircled{1} & 0 & 0 & \textcircled{1} & 0 & 0 \\ 0 & 0 & \textcircled{1} & 0 & 0 & \textcircled{1} & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \end{pmatrix} \tag{10.38}$$

Figure 10.20 displays the Tanner graph of the matrix described in Eq. (10.38).

Tanner Graph can have cycles; in other words, we can start from a variable node and come back to it by following different paths. The size of the smallest cycle is called *girth* of the LDPC matrix. In Fig. 10.18 the matrix has girth 4 and the cycle is shown with the bold red path; the corresponding 1s in Eq. (10.38) are highlighted with a red circle, and they are the vertices of a rectangle.

Cycles are very dangerous in LDPC decoding because it is there where the decoder can be "trapped", being unable to find a solution.

While, conceptually, the encoder is a multiplication between the transmitted data and the generator matrix G, LDPC codes can be effectively decoded by the iterative *Belief Propagation* (BP) algorithm (also known as *Sum-Product* or *SPA*). BP decoding matches the underlying code bipartite graph: decoding message is computed on each variable node and check node, and iteratively exchanged through the edges between neighboring nodes (Fig. 10.21). At the end of every iteration an estimated codeword is produced; by multiplying this temporary codeword with H,

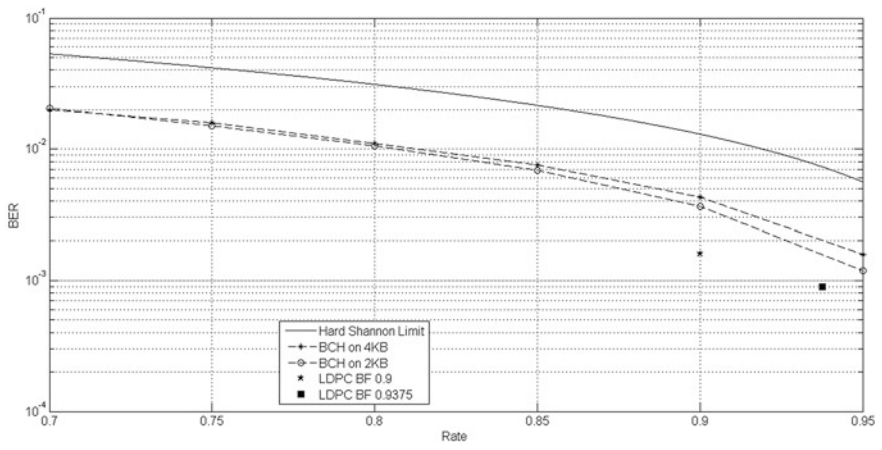

Fig. 10.18 Hard LDPC versus BCH, and hard Shannon limit

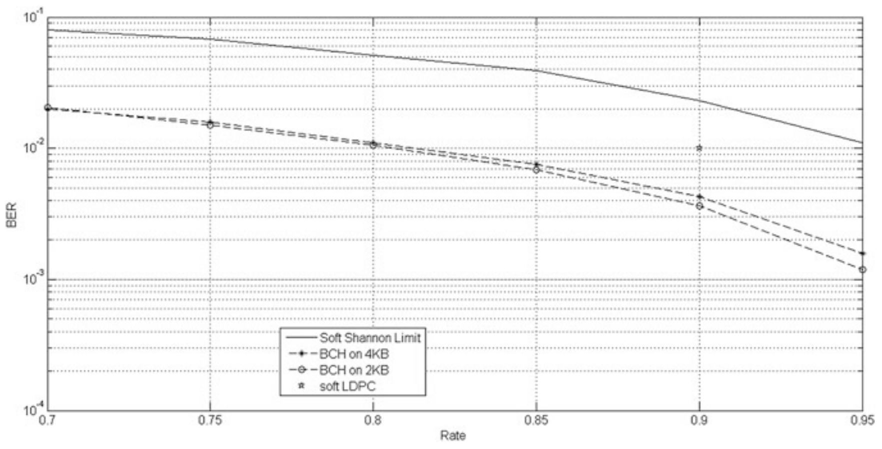

Fig. 10.19 Soft LDPC versus BCH code, and soft Shannon limit

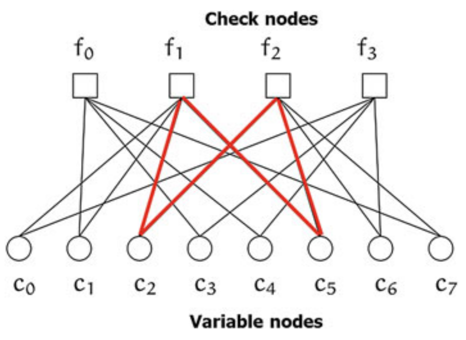

Fig. 10.20 Tanner graph of matrix H of Eq. (10.38)

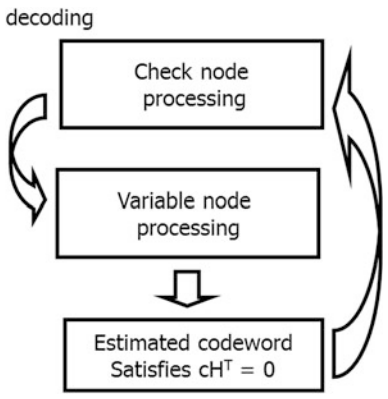

Fig. 10.21 Iterative LDPC decoding

we can check if it is a correct one. If this is the case, then decoding stops, otherwise a new iteration starts. It is well known that BP decoding algorithm works well if the underlying code bipartite graph does not contain too many short cycles. Thus, it is typically required that the graph is 4-cycle free, which is relatively easy to achieve. The construction of graphs with higher order cycle free is definitely not trivial.

There a lot of different LDPC families. The LDPC code is called (j, k)-regular if each variable node has a degree of j and each check node has a degree of k. There are also irregular codes. To be useful for Flash memories, LDPC codes must not only achieve very low decoding error rate with high Code Rates, but also be suitable for high-speed VLSI implementation, with minimal silicon and energy cost. It has been well demonstrated that *Quasi-Cyclic* (QC) LDPC codes are one family of such implementation-oriented LDPC codes. The parity check matrix of a QC-LDPC code consists of arrays of circulants. A circulant is a square matrix in which each row is the cyclic shift of the row above it, and the first row is the cyclic shift of the last row. The parity check matrix H of a QC-LDPC code can be written as

$$H = \begin{bmatrix} H_{1,1} & H_{1,2} & \cdots & H_{1,n} \\ H_{2,1} & H_{2,2} & \cdots & H_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ H_{m,1} & H_{m,2} & \cdots & H_{m,n} \end{bmatrix} \tag{10.39}$$

where each sub-matrix $H_{i,j}$ is a binary circulant. Data storage systems such as Flash demand very high Code Rates (e.g. 8/9 and higher). It has been proved that LDPCs with best performances are the irregular ones [16–18]. However, with high Code Rates, regular QC-LDPC codes are typically used, because they are easier to implement in hardware. In this case, all the rows have the same number of 1s, all the columns have the same number of 1s, and all the sub-matrices $H_{i,j}$ have the same column weight of 1 or 2. Since LDPC codes are subject to error floor, the code parity check matrix column weight is typically 4, or even higher, in order to ensure a sufficiently low error floor (e.g., error floor only occurs below the decoding failure rate of 10^{-12}) [5]. The regular and cyclic structure of QC-LDPC code parity check matrix can be leveraged to largely improve its encoder and decoder implementation efficiency as described below.

10.3.1 LDPC Codes and NAND Flash Memories

Planar TLC NAND has recently pushed for LDPC codes adoption, mainly because of the very high NAND raw BER. The complexity for tuning LDPC codes to the NAND characteristics is definitely high. The good thing is that the industry already paid the price (in terms of R&D) and today LDPC can be leveraged to foster the 3D evolution (shrink) even more.

A Read operation in the NAND environment is of a hard type by its nature. Sense Amplifiers translate cells threshold voltages into digital values, "0" or "1" (Chap. 3). This is the reason why it is not easy to extract a soft information.

In Fig. 10.22, the two V_{TH} distributions represent the two possible cell states: "0" and "1" (assuming SLC NAND). When distributions overlap, errors pop up. A hard decision decoder reads all the positive values as 0 and the negative ones as 1, so that the overlap area in the figure represents the NAND raw BER. However, A and B are very different errors, because A is a little positive, while B is far away from 0. It's like saying that B is much more likely to be an error than A. By exploiting the exact value of A and B, the decoder can have a better starting point. This is the so called *soft information* and it is measured by the *Log Likelihood Ratio* (LLR).

The LLR for a particular value x is the logarithmic ratio between the probability that the bit x was a 0 given the read value y, and the probability that the bit x was a 1 given the read value y. Given this definition, LLR can be written as:

$$L(u_i) = \log\left[\frac{P(u_i = 0|y)}{P(u_i = 1|y)}\right] \tag{10.40}$$

With NAND it's not possible to know the exact value of the threshold voltage V_{TH}. As an approximation, each overlap area is split in a number of slices, by moving the reference voltages. Figure 10.23 shows a MLC NAND where each overlap area is split in 4 slices, so that each bit (LSB and MSB) is read with 3 soft bits. The higher the number of soft bits, the more accurate the information is. This technique has a cost because each bit has to be read 3 times (in this example). Basically, soft information is asking for read oversampling.

In order to maximize the return on soft information, it is necessary to carefully understand how to move each read reference voltage, and how many times, since each additional read increases the latency.

The interaction between LDPC and NAND Flash is illustrated in Fig. 10.24.

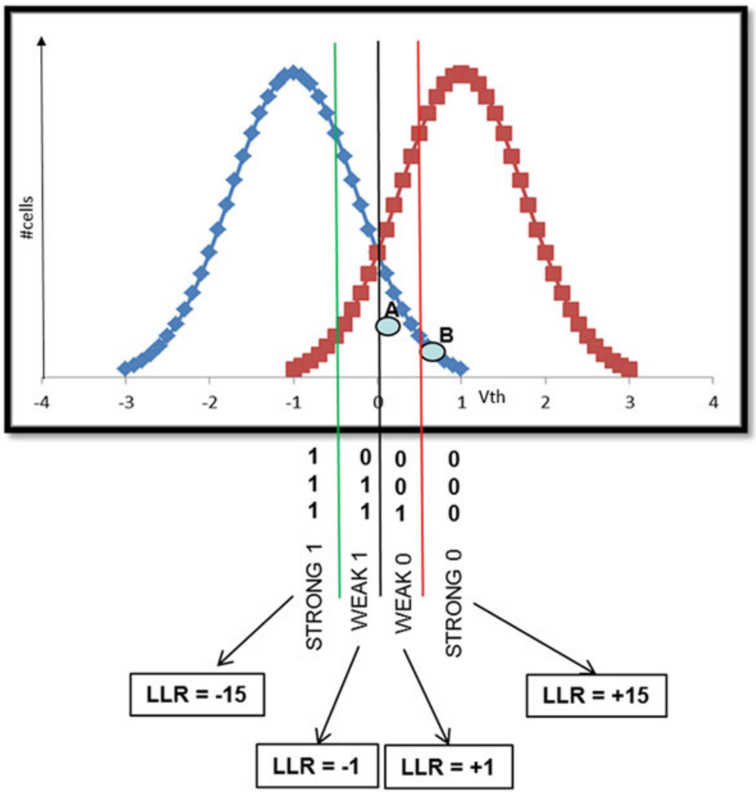

Fig. 10.22 Threshold voltage distributions in SLC NAND flash

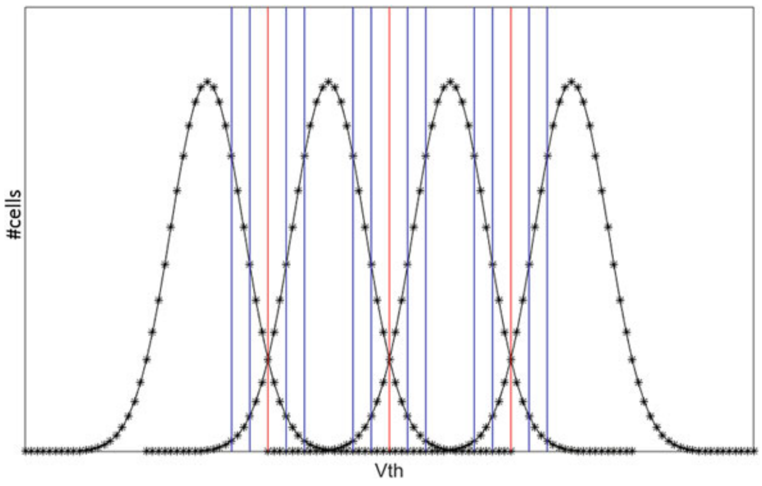

Fig. 10.23 Soft reads in MLC NAND flash

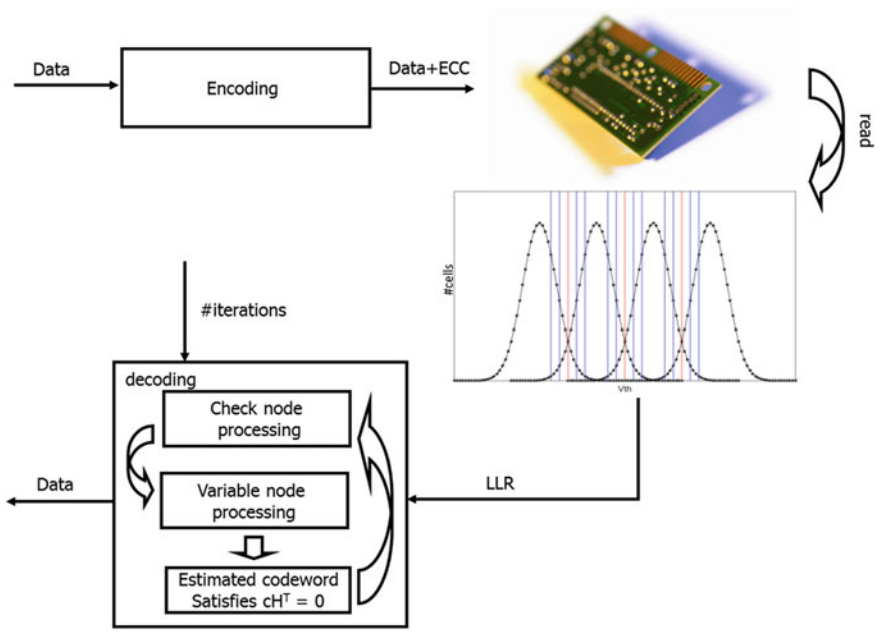

Fig. 10.24 Soft LDPC in the NAND context

10.3.2 LDPC Code Encoding

In the context of LDPC encoder design, the most straightforward approach is to multiply the information bits with the dense generator matrix derived from the sparse parity check matrix. The density of the generator matrix together with a large code length make the parallel implementation of generator matrix-vector multiplication impractical due to very high implementation complexity [19]. Hence, a partially parallel encoder implementation is a must. However, for general non-QC LDPC codes randomly constructed, their dense generator matrices may not have any structural regularity that can be used to develop efficient partially parallel encoder architecture. For QC-LDPC codes, partially parallel encoder design becomes much more affordable. Let's assume that the QC-LDPC code parity check matrix is a $m \times n$ array of circulants, and each circulant is $p \times p$. In the simplest scenario, the matrix has a full rank of $m \cdot p$. We assume that code parity check matrix can be column-wise permuted so that the following sub-array has a full rank of $m \cdot p$:

$$\begin{bmatrix} H_{1,n-m+1} & H_{1,n-m+2} & \cdots & H_{1,n} \\ H_{2,n-m+1} & H_{2,n-m+2} & \cdots & H_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ H_{m,n-m+1} & H_{m,n-m+2} & \cdots & H_{m,n} \end{bmatrix} \tag{10.41}$$

Let's also consider a systematic encoding, i.e. the first $(n - m) \cdot p$ bits in each codeword are the information bits, and the first $(n - m) \cdot p$ columns of the parity check matrix correspond to the $(n - m) \cdot p$ information bits. Hence, the corresponding generator matrix has the following form:

$$G = \begin{bmatrix} I & O & \cdots & O & G_{1,1} & G_{1,2} & \cdots & G_{1,m} \\ O & I & \cdots & O & G_{2,1} & G_{2,2} & \cdots & G_{2,m} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ O & O & \cdots & I & G_{n-m,1} & G_{n-m,2} & \cdots & G_{n-m,m} \end{bmatrix} \tag{10.42}$$

where I and O represent identity $p \times p$ matrix and zero $p \times p$ matrix. Being G the generator matrix, it must satisfies $H \cdot G^T = 0$, which clearly suggests that each $G_{i,j}$ should also be a $p \times p$ circulant.

The generator matrix-vector multiplication for QC-LDPC encoding can be carried out in a partially parallel manner by leveraging the inherent cyclic structure of the generator matrix (Fig. 10.25).

If the matrix H is not full rank, the code is semi-systematic. In other words, the matrix G is shown in Eq. (10.43),

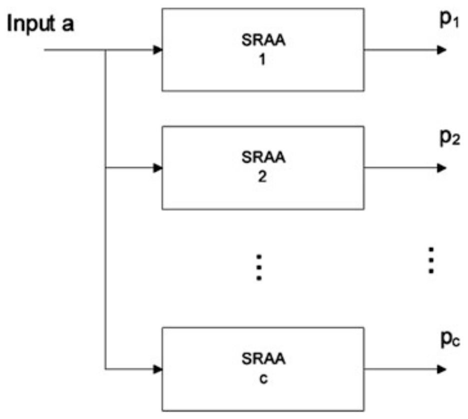

Fig. 10.25 LDPC encoding with a full-rank matrix

$$G = \begin{bmatrix} I & & & & G_{1,1} & \cdots & G_{1,z} \\ & I & & & G_{2,1} & \cdots & G_{2,z} \\ & & \ddots & & \vdots & & \vdots \\ & & & I & G_{n-z,1} & \cdots & G_{n-z,z} \\ 0 & 0 & \cdots & 0 & Q_{1,1} & \cdots & Q_{1,z} \\ \vdots & & & \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & 0 & Q_{z,1} & \cdots & Q_{z,z} \end{bmatrix} \tag{10.43}$$

where the part represented by Q is neither systematic nor regular (in size).

The hardware structure is represented in Fig. 10.26. The systematic part is equivalent to the one of the full-rank H matrix. The grey part is non-systematic and is not regular since the size of the Qs circulants is not fixed. In addition to that, it is not easy to make it parallel due to its irregularity.

During read, once decoding stops, it is necessary to multiply the non-systematic part by Q^{-1}, in order to recover the original data [20].

As discussed, a semi-systematic implementation is much more complex than a systematic one. When H is not full-rank, a possible workaround is to fix the parity section. Parity-check matrix H on the parity section is composed by a specific circulants. Those circulants can be all-zeros circulants so that matrix H won't be regular anymore. For more detailed discussions on QC-LDPC code encoder design, readers can refer to [19, 21, 22].

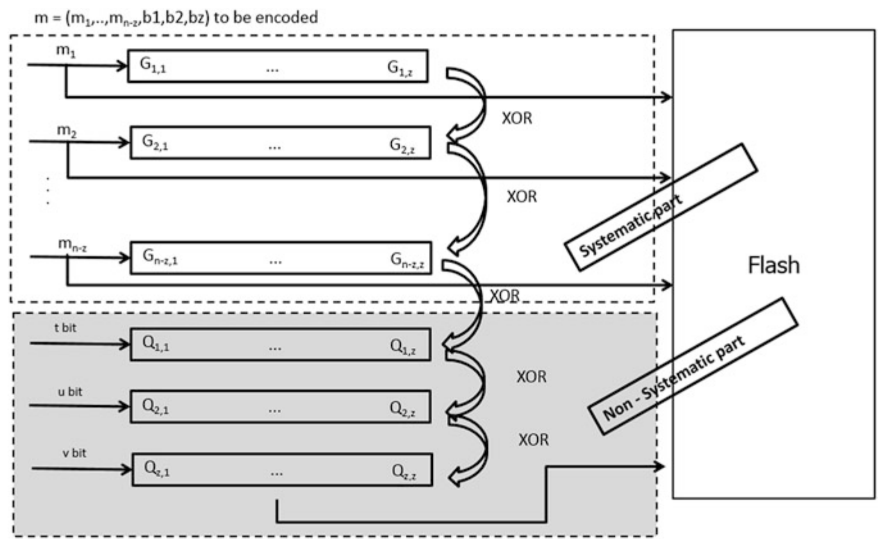

Fig. 10.26 LDPC encoding without a full-rank matrix

10.3.3 LDPC Code Decoding

To understand LDPC decoding, one of the key concepts is the extrinsic information. Here it is explained through an example [5].

We have a troop of 6 soldiers and each soldier wants to know the total number of soldiers in the troop. In Fig. 10.27 we have a linear troop. In this case, each soldier takes the number provided by the neighbour behind, he adds 1, and he transmits the result to the neighbour in front of him. Soldiers at the edges receive a 0 from the side without neighbour. For each soldier, the sum of received and transmitted numbers is equal to the total number of soldiers.

The second troop (Fig. 10.28) is a little more complex, and it requires different rules to pass the information. Each soldier takes all the numbers from his neighbours, he adds 1, and he subtracts the number passed by the neighbour he wants to send the message to. For example, the yellow soldier sends 2 + 3 + 2 + 1 − 2 = 6 to the green soldier. Soldiers at the edges receive a 0 from the side without neighbour. The sum of the number that a soldier receives from anyone of his neighbours plus the one the soldier passes to that neighbour is equal to the total number of soldiers. This introduces the concept of *extrinsic information*. The idea is that a soldier does not pass to a neighbouring soldier any information that the

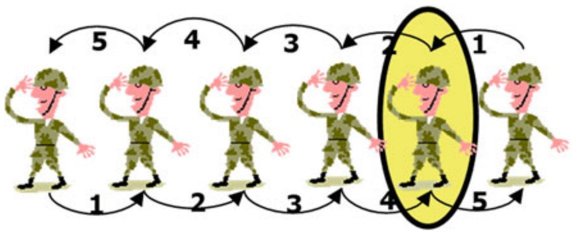

Fig. 10.27 Linear troop

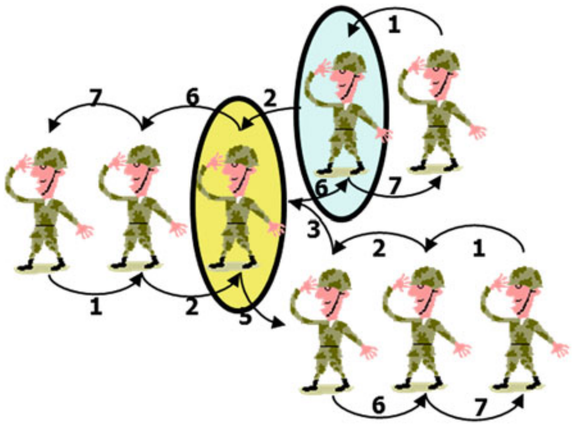

Fig. 10.28 Extrinsic information

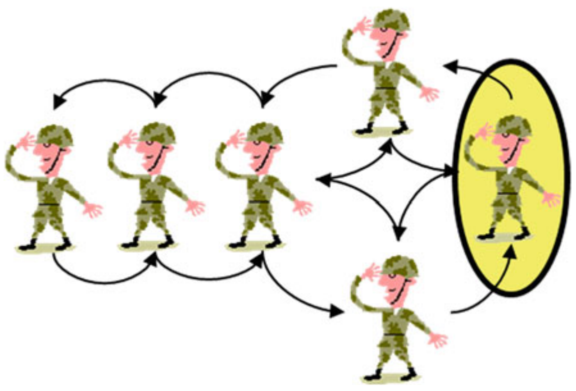
Fig. 10.29 Soldier formation containing a cycle

neighbouring soldier already has; in other words only extrinsic information is passed.

The last troop (Fig. 10.29) contains a cycle. The situation is unsolvable: no matter what counting rule one may devise, the cycle represents a type of positive feedback, both in clockwise and counter-clockwise direction, so that the messages passed within the cycle will increase without bound. This shows that the message passing on a graph cannot be claimed to be optimal if the graph contains one or more cycles. However, while most practical codes contain cycles, it is well known that message-passing decoding performs very well, assuming properly designed codes.

The key innovation behind LDPC codes is the low-density nature of the parity check matrix, which facilitates iterative decoding. Message-passing decoding refers to a collection of low-complexity decoders working in a distributed fashion to decode a received codeword, in a concatenated coded scheme. We can better understand this sentence by using the crossword-puzzle analogy (Fig. 10.30).

Solving a crossword-puzzle proceeds as follows:

- start with all the horizontal words we know → red circles;
- proceed with all the vertical words we know → blue circles;
- re-start to see if we are able to complete more horizontal words given the addition of the vertical words of the previous step → green circles;
- re-start to see if we are able to complete more vertical words → magenta circles;
- keep looping until the crossword-puzzle is completed (or a codeword is found) and stop when either we are not able to solve it (we fell in error floor) or we are too tired (we reached the maximum number of iterations).

Belief Propagation algorithm is the best iterative decoding methods for LDPC. In order to understand it, it might be useful to consider the Tanner Graph of the parity check matrix (Fig. 10.31).

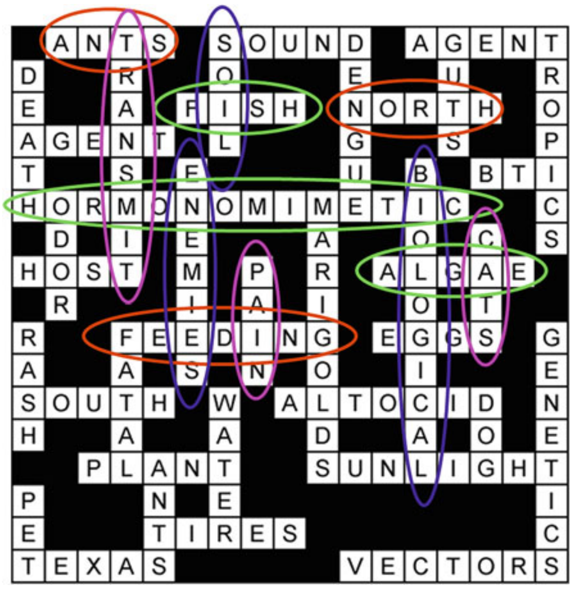

Fig. 10.30 Crossword-puzzle

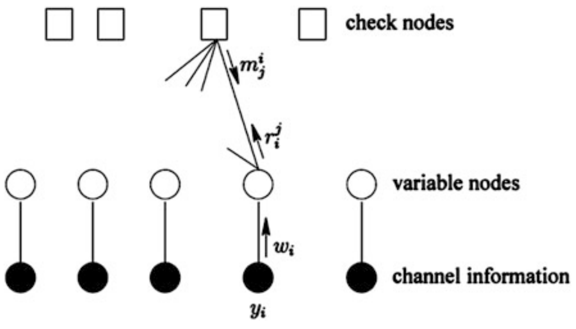

Fig. 10.31 Tanner graph of a LDPC parity check matrix

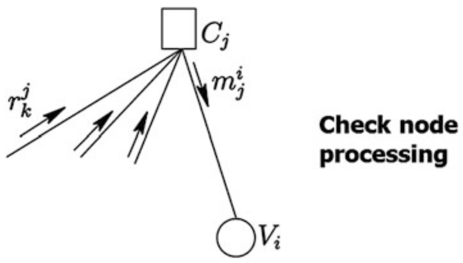

Fig. 10.32 Check node processing of LDPC BP decoding

During the check node processing phase (Fig. 10.32), each check node has to compute the values m it has to send to variable nodes it is connected to. Values are computed according to Eq. (10.44).

$$m_j^i = \prod_{k \in N(j)\backslash\{i\}} sign\left(r_k^j\right) \cdot \phi\left(\sum_{k \in N(j)\backslash\{i\}} \phi\left(\left|r_k^j\right|\right)\right) \tag{10.44}$$

$$\phi(x) = -\log\left(\tanh\left(\frac{x}{2}\right)\right) \tag{10.45}$$

Remembering the soldier example, please note that only the extrinsic information is taken into account: in fact, the value m_i is computed by using all the values sent by the variable nodes connected to that specific check node, except variable node i.

The same idea applies to variable node processing (Fig. 10.33), where value r_j is computed by using all the values sent by the check nodes connected to the variable node, except check node j. Equation (10.46) is used

$$r_i^j = w_i + \sum_{k \in N(i)\backslash\{j\}} m_k^i \tag{10.46}$$

where w are the input LLRs.

Values r represent the estimated codeword. At the end of each iteration this word is multiplied by the transpose of H to check if it is a real codeword. If the result is null, then r is a codeword and the decoding is finished, otherwise a new iteration starts.

The formula used for check node processing is a very complex one and it involves the function tanh, which is sketched in Fig. 10.34.

BP can be approximated with the so-called *min-sum* decoding algorithm: the computational complexity can be largely reduced by paying a small decoding

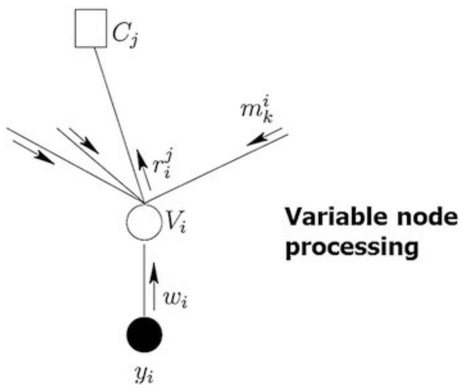

Fig. 10.33 Variable node processing of LDPC BP decoding

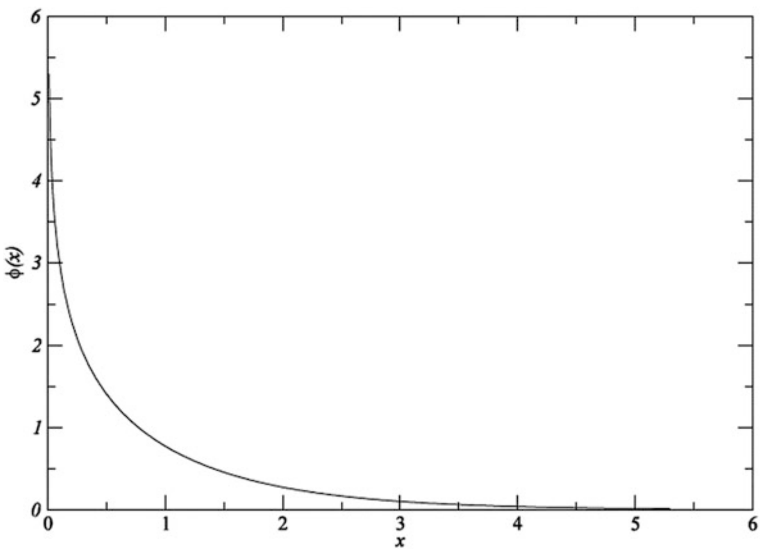

Fig. 10.34 Tanh function

performance degradation. The main difference between BP and min-sum lies in the check node: Eq. (10.44) applies to BP, while the check node processing for min-sum is described by Eq. (10.47).

$$m_j^i = \prod_{k \in N(j) \setminus \{i\}} sign\left(r_k^j\right) \cdot \min_{k \in N(j) \setminus \{i\}} \left|r_k^j\right| \tag{10.47}$$

Therefore, the function $\Phi(x)$ (i.e. tanh), which is typically implemented as LUT, is eliminated in the min-sum decoding algorithm. Min-sum can be further optimized, as described below.

Figure 10.35a shows the comparison between values computed via sum-product (SPA) and values computed via min-sum. Dots on the bisector would mean that min-sum is a great approximation of sum-product, but this is not the case; even the average has a different slope. By introducing an attenuation factor α, the approximation can be much better, as shown in Fig. 10.35b.

In other words, Eq. (10.47) can be computed as

$$m_j^i = \alpha \cdot \prod_{k \in N(j) \setminus \{i\}} sign\left(r_k^j\right) \cdot \min_{k \in N(j) \setminus \{i\}} \left|r_k^j\right| \tag{10.48}$$

The attenuation factors could change at each iteration and they must be properly studied.

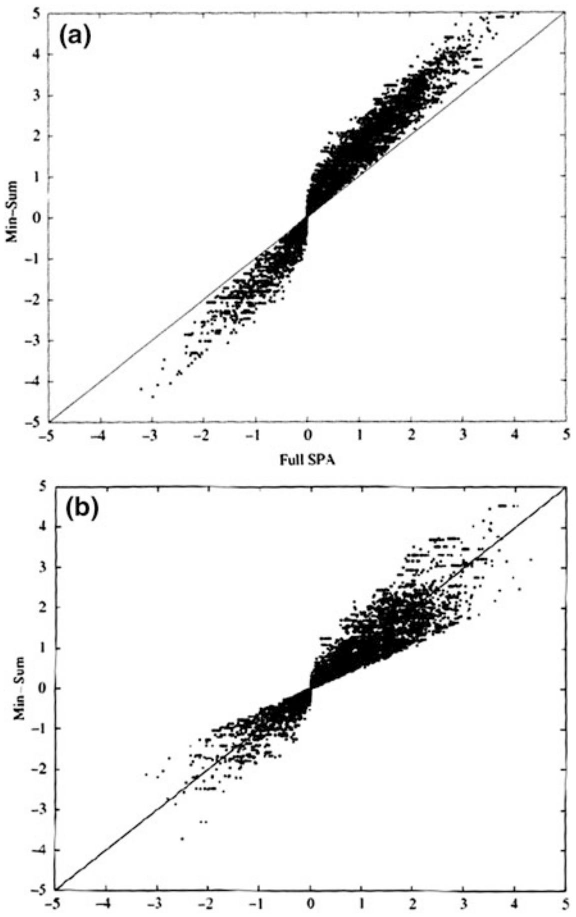

Fig. 10.35 **a** Comparison between check node variables computed with SPA and min-sum. **b** Comparison between check node variables computed with SPA and normalized min-sum [25]

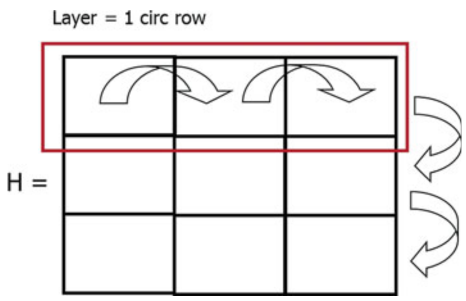

Fig. 10.36 LDPC layered decoding

Regardless of the specific decoding algorithm, the hardware implementation can be parallelized by splitting the circulants processing (for both variable and check nodes), as sketched in Fig. 10.36. This solution is known as “layered decoding”.

Taking again the crossword-puzzle analogy, in the min-sum case we first work on all horizontal words (check nodes) and only then we switch to the vertical words (variable nodes). In the layered case, once we have enough information on the horizontal words (check nodes of one circulant row), we immediately switch to the vertical words (variable nodes). In this way, the computation on the check nodes of the second layer (second circulant row) has a much cleaner input (because it doesn’t use the initial variable node value but the one already computed by the first layer). Indeed, layered decoding requires much less iterations than standard min-sum.

10.3.4 QC-LDPC Applied to NAND Flash Memories

For Enterprise SSDs, target UBER is 10^{-16} (Eq. 10.3). Unfortunately, it is not possible to evaluate LDPC performances without simulations, since there aren’t any closed formulas like in the BCH case.

In addition to that, LDPC decoding algorithm, because of its iterative nature, has a big drawback known as *error floor* [23, 24, 27].

Figure 10.37 shows how the error floor manifests itself: it is basically a change of the slope at low BER. With BCH it is possible to exactly predict at which BER the resulting UBER will be 10^{-16}; with LDPC we don’t know at which BER the error floor will appear and its slope. The only certainty is that it will appear.

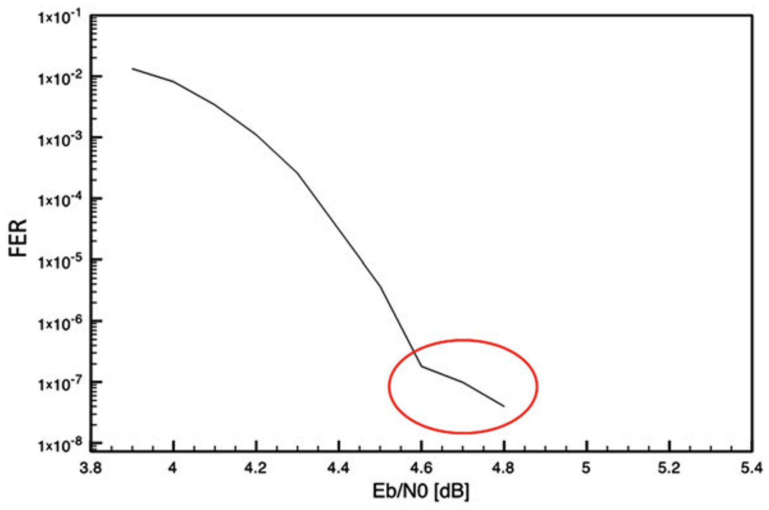

Fig. 10.37 Error floor

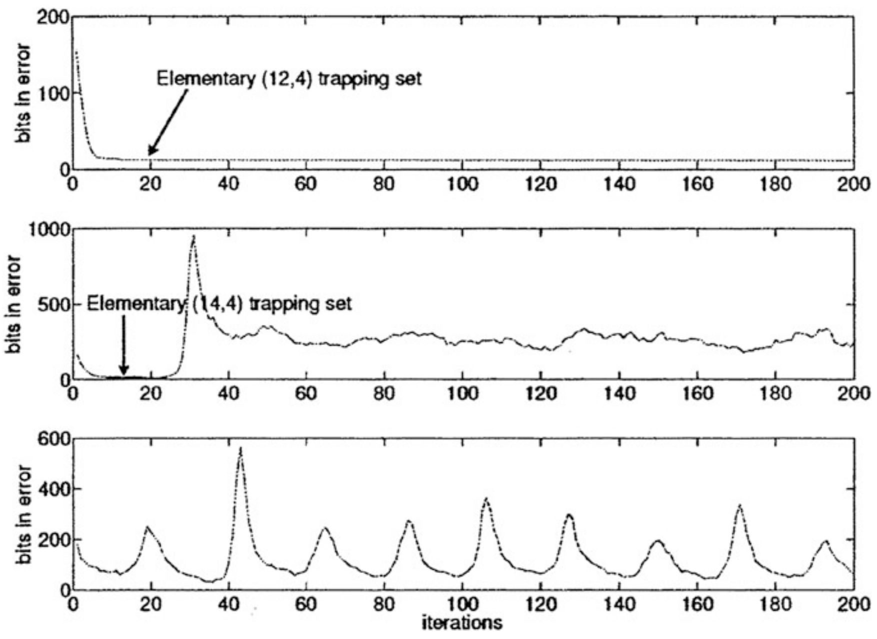

Fig. 10.38 Different kind of trapping sets [26]

It is still a mystery why error floor pops up. Nowadays, mathematicians think that it is due to *trapping sets*. Once the decoder is trapped in a trapping set, values of the variable nodes corresponding to some of the wrong bits become bigger and bigger as decoding proceeds; in other words, at some point, it becomes almost impossible for the decoder to revert its decision. The decoding will reach the maximum number of allowed iterations without finding a codeword.

Because there are 3 different types of trapping sets (Fig. 10.38), the output of the decoder might be:

- a codeword containing few constant errors;
- a codeword containing a random number of errors;
- a codeword that contains a periodical number of errors.

The last one is very dangerous because a codeword with 6 errors can have 200 errors after decoding!

Going back to the simulation topic, software simulations are not a viable solution to reach a UBER of 10^{-16}; hardware co-simulations are a must. A single FPGA can run few hundred million codewords per day, and this acceptable only if the target FER is in the range of 10^{-6}.

On the other hand, because of error floor, it is not possible to approximate the graph below FER of 10^{-6} with a simple straight line. Bottom line, enterprise applications ask for simulations of not less than 10^{13} codewords. One FPGA would

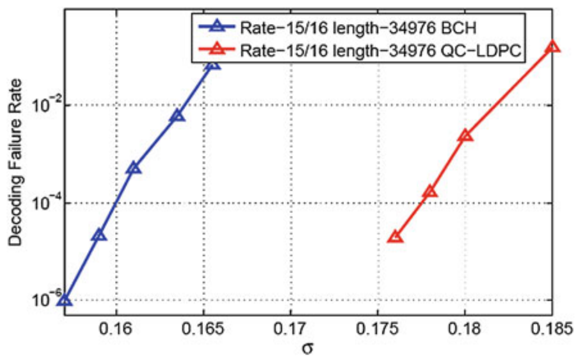

Fig. 10.39 QC-LDPC versus BCH for 2-bit/cell NAND flash memory [12]

need 100,000 days of simulations! This is why networks of FPGAs are the only practical solution to this problem [24].

It is worth highlighting that it is important to run the “correct” simulations. In fact, each parameter change requires a different simulation. For example, it is not possible to extract the soft error floor from the hard error floor. For the same reason, the min-sum decoding error floor can’t be used to deduce the floor for the normalized min-sum.

Figure 10.39 shows a comparison between LDPC and BCH on AWGN channel. NAND V_{TH} distributions are modeled as two symmetric Gaussian distributions, whose mean values are $V_{TH} = -1$ and $V_{TH} = +1$, respectively. In this model the NAND raw BER is represented by the variance σ of the distributions.

In order to understand the actual performances of a specific LDPC code, it is fundamental to make simulations based on data extracted from silicon.

Data read from NAND Flash memories are always either a 0 or a 1, as already explained in Sect. 10.3.2. Therefore, the starting point is always hard decoding; if it fails, soft decoding takes over and we need to:

- Re-Read in order to get reliability info for each single bit;
- Map each bit to a LLR value;
- run soft simulations.

Re-Read strategy is described in Sect. 10.3.2: basically, the read reference voltage is shifted, and one or more additional Read operations are performed to understand where bits are located within the voltage distribution.

Table 10.1 Example of LLR values for soft decoding

Value read from NAND flash		
1st read	2nd read (re-read)	LLR
0	0	+7
0	1	+1
1	0	−1
1	1	−7

Each Re-Read operation returns a sequence of 0s and 1s, which can be coupled to the sequence of the previous Read, as shown in Table 10.1.

The LLR sign indicates whether the bit of the 1st Read is more likely to be a 0 or a 1; the magnitude indicates the confidence level associated to the 1st Read. Let's look at a couple of examples: "+1" indicates that we have read a 0 but we are not that confident, while "+7" indicates that we have read a 0 and we are pretty sure about this bit to be correct.

Once each bit of the transmitted message has been mapped to an LLR value, this value is the input for soft decoding simulations, which are used to build curves like the one shown in Fig. 10.19.

To sum up, despite all the challenges related to error floor and soft information, LDPC can successfully be utilized to boost ECC performances, and it is definitely the most promising solution for 3D NAND Flash memories, especially when looking at TLC and QLC storage.

References

1. C.E. Shannon, A mathematical theory of communication. Bell Syst. Tech. J. **27**(379–423), 623–656 (1948)
2. C. Berrou, A. Glavieux, P. Thitimajshima, Near Shannon limit error-correcting coding and decoding: Turbo-codes, in *Proceedings of ICC'93*, Geneve, Switzerland, May 1993, pp. 1064–1070
3. R. Micheloni, A. Marelli, K. Eshghi, *Inside Solid State Drives (SSD)* (Springer, Berlin, 2012)
4. R. Micheloni, A. Marelli, R. Ravasio, *Error Correction Codes for Non-Volatile Memories* (Springer, Berlin, 2008)
5. S. Lin, D.J. Costello, *Error Control Coding* (Prentice Hall, Upper Saddle River, 2004)
6. T.K. Moon, *Error Correcting Coding—Mathematical Methods and Algorithm* (Wiley, NJ, 2005)
7. R.C. Bose, D.K. Ray-Chaudhuri, On a class of error correcting binary group codes. Inf. Control **3**, 68-79 (1960)
8. A. Hocquengheim, Codes Correcteurs d'erreurs. Chiffres 2, Sept 1959
9. M.A. Pellegrini, The (2, 3)-generation of the classical simple groups of dimensions 6 and 7. Bull. Aust. Math. Soc. **93**(1), 61–72 (2016)
10. M.A. Pellegrini, M.C. T. Bellani, The simple classical groups of dimension less than 6 which are (2, 3)-generated. J. Algebra Appl. **14**(10), 1550148 (2015) (15p)
11. M. Kim et al. Decoder error probability of binary linear block codes and its application to binary primitive BCH codes. IEICE Trans. Fundam. (1996)
12. R. Micheloni, L. Crippa, A. Marelli, *Inside NAND Flash Memories* (Springer, Berlin, 2010)
13. E.R. Berlekamp, *Algebraic Coding Theory* (McGraw Hill, New York, 1968)
14. H.O. Burton, Inversionless decoding of binary BCH codes. IEEE Trans. Inf. Theory **17** (1971)
15. Y. Lee, H. Yoo, I. Yoo, I.C. Park, 6.4 Gb/s multi-threaded BCH encoder and decoder for multichannel SSD controllers, in *ISCC Digest of Technical Papers* (2012)
16. R.G. Gallager, Low-density parity-check codes. IRE Trans. Inf. Theory **IT-8**, 21–28 (1962)
17. V. Zyablov, M. Pinsker, Estimation of the error-correction complexity of Gallager low-density codes. Problemy Peredachi Informatsii **11**, 23–26 (1975)
18. R.M. Tanner, A recursive approach to low complexity codes. IEEE Trans. Inf. Theory **IT-27** (5), 533–547 (1981)

19. G.A. Margulis, Explicit constructions of graphs without short cycles and low density codes. Combinatorica **2**(1), 71–78 (1982)
20. Z. Li et al., Efficient encoding of quasi-cyclic low-density parity check codes. IEEE Trans. Commun. (2006)
21. S. Myung et al., Quasi-cyclic LDPC codes for fast encoding. IEEE Inf. Theory June (2005)
22. Z. Li, L. Chen, S. Lin, W. Fong, P.-S. Yeh, Efficient encoding of quasi-cyclic low-density parity-check codes. IEEE Trans. Commun. **54**, 71–81 (2006)
23. T. Richardson, Error floors of LDPC codes, in *Proceedings of the 41st Annual Allerton Conference on Communication*, USA (2003)
24. R. Micheloni et al., Hardware/software co-simulation for error-floor detection in LDPC, in *Proceedings of Flash Memory Summit*, www.flashmemorysummit.com, Santa Clara, CA, USA, 5–7 Aug 2014
25. M. Fossorier et al., Channel Coding: Theory, Algorithms and Application (Wiley, NJ, 2005)
26. S. Landner, O. Milenkovic, Algorithmic and combinatorial analysis of trapping sets in structured LDPC codes, in *International Conference on Wireless Networks* (2005)
27. L. Dolecek et al., Predicting error floors of structured LDPC codes: deterministic bounds and estimates. IEEE J. Sel. Areas Commun. (2009)

Chapter 11
Advanced Algebraic and Graph-Based ECC Schemes for Modern NVMs

Frederic Sala, Clayton Schoeny and Lara Dolecek

In this chapter, we discuss advanced error-correcting code techniques. In particular, we focus on two complementary strategies, asymmetric algebraic codes and non-binary low-density parity-check (LDPC) codes. Both of these techniques are inspired by traditional coding theory; however, in both cases, we depart from classical approaches and develop new concepts specifically designed to take advantage of inherent channel characteristics that describe non-volatile memories.

We focus, in particular, on modern flash devices, including multi-level and 3D flash technology. Flash is a phenomenally popular technology; the attention it has received has led to numerous process innovations. As a result, current implementations of flash, such as 3D flash, contain vast numbers of tightly-packed transistors. Flash cells suffer from a variety of physical issues, including interference/crosstalk (stronger in certain dimensions compared to others due to the packing design parameters in 3D flash), read and write disturbs, charge leakage, and many others. These complex effects are poorly modeled by traditional channels and the resulting errors are not well handled by traditional coding schemes; we must look towards novel approaches. We select two distinct, opposite points of attack. The first is improving on classical algebraic codes, which offer known, efficient encoding and decoding algorithms and are suitable for inexpensive, efficient devices with mild error tolerance requirements. The second is improving on cutting-edge non-binary LDPC codes, which have among the very best error-correcting ability of all known coding schemes, at the cost of more complex encoding and decoding circuitry. Additionally, new algebraic codes are particularly suitable for hard-read channels

F. Sala · C. Schoeny · L. Dolecek (✉)
Electrical Engineering Department, UCLA, Los Angeles, CA 90095, USA
e-mail: dolecek@ee.ucla.edu

F. Sala
e-mail: fredsala@ucla.edu

C. Schoeny
e-mail: cschoeny@ucla.edu

R. Micheloni (ed.), *3D Flash Memories*, DOI 10.1007/978-94-017-7512-0_11

whereas LDPC codes are most beneficial for soft-read channels. The two coding approaches thus target the opposite ends of the flash quality/cost tradeoff curve.

In the case of algebraic codes, we discuss a set of code constructions which rely on traditional symmetric codes, such as BCH codes, as building blocks. The final result is a family of codes specifically tailored towards asymmetric channels, such as the triple-level cell (TLC) flash data storage channel, which can be deployed in both 2D and 3D flash. We introduce a variation of these codes which can handle a type of very specific flash errors, along with codes suitable for the dynamic thresholding scheme, which is effective for non-volatile memories. For a subset of our techniques, we quantify the offered improvements on data sets measured from real flash devices.

In the case of LDPC codes, we present design and optimization techniques that result in non-binary LDPC codes with lowered error floors. The error floor is an effect which reduces the improvement in the output error rate of iteratively-decoded LDPC codes as the input SNR increases; this effect occurs for high SNRs, limiting the applicability of LDPC codes for high-reliability applications, such as flash. In order to resolve this problem, we identify certain subgraph objects, called absorbing sets, which occur in the Tanner graph structure of the LDPC code and contribute to the error floor. We characterize these objects for the non-binary LDPC case and present an algorithm to remove the smallest absorbing sets. Here too, the resulting code construction is tailored for an asymmetric channel. The power of the technique is illustrated for a series of non-binary LDPC codes, including the practical quasi-cyclic (QC-LPDC) codes.

11.1 Asymmetric Algebraic ECCs

One of the most interesting features of real-life memory channels is their asymmetry; that is, the fact that not all errors in such channels occur with equal probability. For example, the channel induced by a multiple-level flash storage device has a vastly higher chance of inducing an error between the erased state and a non-erased state compared to that of errors between two non-erased states.

Traditional coding theory largely does not concern itself with these asymmetries. The binary symmetric (BSC) and binary erasure (BEC) channels are the most frequently studied discrete channels while the additive white Gaussian noise (AWGN) channel is the most used continuous channel. None of these channels model asymmetries beyond the particular channel parameters. As a result, in order to apply tools from traditional coding theory to real-life situations, a symmetric channel is selected based on the worst-case error. This conservative approach allows for a safe margin.

On the other hand, such approaches are also wasteful for asymmetric channels, since a large amount of the strength of the code is then applied towards correcting errors which are rare. This unneeded strength results in a lower-than-necessary code rate, wasting energy or storage capacity. Conversely, if the code rate is kept

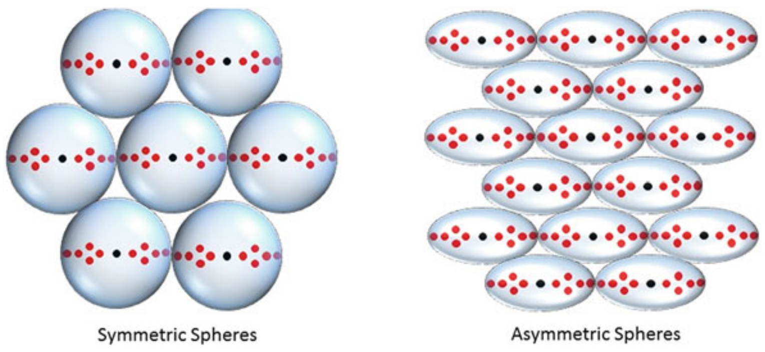

Fig. 11.1 The *left diagram* represents a packing of traditional Hamming spheres that are agnostic to the error distribution. The *right diagram* represents a packing of spheres that are designed with the error asymmetry in mind. The *black dot* at the center of each sphere represents the codeword, and the *red dots* represent the most likely erroneously received words. By targeting the specific error distribution, we can pack more asymmetric spheres than symmetric spheres, which translates into a higher code rate. Note that this is a simplified illustration of n-dimensional spheres

constant, it would be more effective to place the code's power into correcting frequent errors, thus improving the overall error probability of the system. This concept of coding for asymmetries is illustrated in Fig. 11.1.

In the remainder of this section, we discuss asymmetric error-correcting codes. We formalize the intuition presented in the previous discussion. As described earlier, we focus especially on the case of data storage in flash. In fact, data sets collected from production flash devices are available. Since encoders and decoders for algebraic codes are easy to specify and implement, we can test our proposed codes directly on the real data (rather than perform simulations using synthetic data).

11.1.1 Graded-Bit-Error Correcting Codes

We begin by considering the TLC (triple-level cell) flash channel, which, until very recently, was the most advanced and dense flash technology. Despite the name given to these devices, each cell has eight possible charge levels and thus represents three bits of information. The organization of flash devices places each of these three bits on a different page; pages are themselves collected as blocks, which are further organized into planes [1].

This organization allows us to model the TLC flash channel in two natural ways. First, looking at each cell separately, we may view the cell as an 8-ary channel, as each cell has eight possible states. Secondly, we may view each bit separately, since these bits are placed on different pages. In this case, the cell can be modeled as three independent binary channels.

In the case of the 8-ary channel, we may apply non-binary codes. We can use the statistics of the 8-ary channel to estimate the number of errors expected in a block of cells. Using this information and a target error rate, we can select an appropriate code, such, as for example, a code from the 8-ary BCH code family. Similarly, if we view the cell as three independent binary channels, we can select three binary codes, such as three binary BCH codes, based on the error probability of each channel.

It turns out, however, that neither of these approaches is suitable. An $[n,k,t]_8$ BCH code (t-error-correcting 8-ary code of length n and dimension k, e.g., containing 8^k codewords) corrects any t 8-ary errors. For example, an error between states 2 and 6 (a $2 \to 6$ error) can be corrected just as well as a $1 \to 2$ error. However, our channel produces vastly more $1 \to 2$ errors. In particular, most errors are only in one bit of the three-bit binary representation of each 8-ary state.

To illustrate this idea, we present the most frequent errors that occurred on a TLC flash chip over 5000 program/erase (P/E) cycles of operation. Comparing the programmed and errored state for the most frequent errors indeed confirms that most errors occur only in a single bit of the three-bit triplet (Table 11.1).

Table 11.1 Most frequent errors measured in a TLC flash device (3-bit cells)

Programmed state	Errored state	Fraction of errors
000	010	0.2467
000	001	0.2444
111	101	0.0820
111	110	0.0807
000	100	0.0669
011	001	0.0556
100	110	0.0550
011	010	0.0547
100	101	0.0540
111	011	0.0217

The left column gives the intended, programmed state of the cell; the middle column shows the result, which contains an error. The right column gives the fraction of total errors caused. Note that each of these 10 most popular errors contain only one bit in error

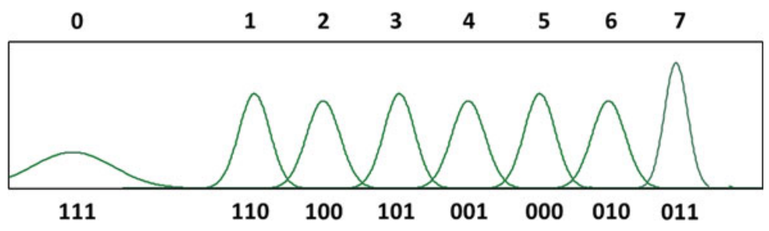

Fig. 11.2 Distributions for different voltage levels in a TLC (3-bit/8-state) cell. The 3-bit representations rely on a Gray code

The reason for this is explained by Fig. 11.2: the 3-bit binary representation of the levels is based on a Gray code, so that going from one consecutive state to the next only changes a single bit. We conclude that the ability to correct many $2 \rightarrow 6$ errors is an inefficiency in the code.

In the binary case, there is a similar problem. As can be predicted from the fact that the three binary channels are really all operating in a single cell, the channels are not independent. In this case, assuming independence underestimates the number of errors where more than one of the bits is in error. That is, errors such as $\mathbf{e} = (1,1,0)$ and $\mathbf{e}' = (1,0,1)$ (here each non-zero value in a triplet represents an error in one of the three bits) are under-represented. In fact, in our TLC device, we measured that the fraction of errors that have 2 bits in error is 0.0314 and the fraction with 3 bits in error is 0.0069. These quantities are far too large to have been produced if the probability of error among each of the pages was independent. Nevertheless, the separate binary code approach is a more accurate model compared to the 8-ary channel.

How can we design a code tailored to specifically deal with such error patterns? First, as shown in the table above, we can profile the channel to discover how many errors are typically single-bit errors and how many are multiple-bit errors. We then seek to introduce a code which corrects errors with precisely these ratios. We detail this notion in the following.

Definition 1 Let $t,v > 0$. Then, a vector $\mathbf{e} = (\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_n)$ over $(GF(2)^m)^n$ is called a [t;v]-**bit error vector** if it satisfies the following two properties:

1. $wt(\mathbf{e}) = |\{i : \mathbf{e}_i \neq 0\}| \leq t$, and
2. for all i, $wt(\mathbf{e}_i) \leq v$.

Definition 2 Let $0 < v_1 < v_2 \leq m$ and $t_1, t_2 > 0$. A vector $\mathbf{e} = (\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_n)$ over $(GF(2)^m)^n$ is a $[t_1, t_2; v_1, v_2]$-**graded bit error vector** if it satisfies the following properties:

1. $wt(\mathbf{e}) = |\{i : \mathbf{e}_i \neq 0\}| \leq t_1 + t_2$,
2. for all i, $wt(\mathbf{e}_i) \leq v_2$, and
3. $|\{i : wt(\mathbf{e}_i) > v_1\}| \leq t_2$.

In the previous definitions, wt() refers to the Hamming weight of the vector (the number of nonzero components in this vector). The basic idea is to introduce a more refined version of the usual definition of error vectors. Rather than simply counting the number of nonzero components, we classify them according to how many bits are in error as well. The first definition is particularly suitable for the case where all errors involve only a small number of bits. The second definition is more flexible: it enables us to profile errors as involving some number of errors in few bits and a (normally smaller) quantity of errors in a larger number of bits. Next, we define codes which are capable of correcting such error patterns:

Definition 3 Let $v, t > 0$. Then, a code C is a [t;v]**-bit error correcting code** if it is capable of correcting every [t;v]-graded bit error vector.

Definition 4 Let $0 < v_1 < v_2 \le m$ and $t_1, t_2 > 0$. Then, a code C is a $[t_1, t_2; v_1, v_2]$-**graded-bit error correcting code** if it is capable of correcting every $[t_1, t_2; v_1, v_2]$-graded bit error vector.

To see how these definitions work (and how they apply to our asymmetric TLC flash channel), consider the following example. We store vectors of length n, where each element is a 3-bit vector. For the sake of this example, we take $n = 7$. Say we store the vector

$$\mathbf{x} = (000\ \ 110\ \ 010\ \ 101\ \ 000\ \ 111\ \ 000).$$

After some time, we read back the stored data as

$$\mathbf{y} = (111\ \ 110\ \ 110\ \ 101\ \ 010\ \ 111\ \ 010).$$

We can conclude that the error vector was

$$\mathbf{e} = (111\ \ 000\ \ 100\ \ 000\ \ 010\ \ 000\ \ 010).$$

We can classify this error vector as a [3,1; 1,3]-graded-bit-error vector. There are a total of $3 + 1 = 4$ cells in error (4 triplets that are not all 0). Of these four, three have only one bit in error, while the remaining has three bits in error. Based on this, we can take $v_1 = 1$, $v_2 = 3$, and $t_1 = 3$, $t_2 = 1$.

Observe how this classification differs from the much more coarse error definition used by BCH codes. In the case of an 8-ary BCH code, we would simply record that there were 4 errors, not distinguishing between the single-bit and multiple-bit errors.

Next, our goal is to introduce graded bit error-correcting code constructions. As we will see, an operation from linear algebra known as the **tensor product** is a crucial ingredient in these constructions. The tensor product is an operation on matrices defined as follows. Let us say that A is a matrix in $R^{m\times n}$ and B is a matrix in $R^{p\times q}$. Then, the tensor product $A \otimes B$ is defined as

$$A \otimes B = \begin{bmatrix} a_{11}B & \cdots & a_{1n}B \\ \vdots & \ddots & \vdots \\ a_{m1}B & \cdots & a_{mn}B \end{bmatrix}.$$

In other words, $A \otimes B$ is an $mp \times nq$ block matrix where each of the elements of A is (scalar) multiplied by the matrix B. This operation has many important properties in mathematics and physics. In coding theory, it was first used by Wolf to produce a construction of [t;v]-bit error correcting codes [2]:

Construction 1 Let C_A be a code with a parity check matrix given by $H_A = H_2 \otimes H_1$, where H_1 is the parity-check matrix of a binary $[m,k_1,v]_2$ code C_1, and H_2 is the parity-check matrix for a $[n,k2,t]_d$ code C_2, where $d = 2^{m-k1}$. Then, C_A is a [t;v]-bit error correcting code.

We can provide a simple example of such a code construction. For C_1, we use the Hamming code $[3,1,1]_2$, which has parity check matrix

$$H_1 = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix}.$$

In other words, we will use as one of our two constituent codes the binary 3-bit repetition code with codewords {000,111}. Next, we can select a different code for C_2. Note that this code, in our case, must be over GF(4), since our final output must be over GF(8), according to our requirements for TLC flash cells. Since we require a code over GF(4), we let α be a primitive element over this finite field. Then, we can take C_2 to be a $[4,2,1]_4$ code, which also corrects one error:

$$H_2 = \begin{bmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & \alpha \end{bmatrix}.$$

Then, it is not hard to see that the resulting matrix is

$$H_A = \begin{bmatrix} 1 & \alpha & \alpha^2 & 0 & 0 & 0 & 1 & \alpha & \alpha^2 & 1 & \alpha & \alpha^2 \\ 0 & 0 & 0 & 1 & \alpha & \alpha^2 & 1 & \alpha & \alpha^2 & \alpha & \alpha^2 & 1 \end{bmatrix}.$$

Of course, it is possible to take the binary image of this GF(4) matrix:

$$H_A = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 \end{bmatrix}.$$

Since H_1 and H_2 are parity-check matrices for single-error correcting codes with the desired properties, we expect C_A (with parity-check matrix H_A) to be a [1;1]-bit error-correcting code. This is indeed the case: we observe that the columns of H_A are all distinct, so that therefore, an error vector with a single bit in error can be corrected. Moreover, if we group the 12-bit long codewords in C_A into 4 groups of 3 bits each, we regain the GF(8) interpretation of the code.

More recently, a construction for the more refined graded-bit-error correcting codes was introduced [3]. This construction relies on the tensor product operation as well; however, the construction is somewhat more sophisticated:

Construction 2 Let C_B be a code with a parity check matrix given by

$$H_B = \begin{bmatrix} H_2 \otimes H_3 \\ H_4 \otimes H_5 \end{bmatrix}.$$

Here, we have C_1 a $[m,k,v_2]_2$ binary code with parity-check matrix H_1. Let $r = m - k$. We take H_1 to be such that the top r_3 rows of H_1 are a parity-check matrix for an $[m,m - r_3,v_1]_2$ code for some $r_3 < r$. This code will be called C_3 (with parity-check matrix H_3). We let H_5 be the submatrix of H_1 including the bottom $r_5 = r - r_3$ rows of H_1. Finally, we let H_2 be the parity-check matrix for a 2^{r_3}-ary $[n, k_2,t_1 + t_2]_d$ code C_2 ($d = 2^{r_3}$) and H_4 to be the parity-check matrix for a 2^{r_5}-ary $[n, k_4,t_2]_f$ code C_4 ($f = 2^{r_5}$).

Then, C_B is a $[t_1,t_2;v_1,v_2]$-graded bit error correcting code of length n.

Let us see how the decoding works for this type of code. For the code C_B, we introduce the decoder D_B which takes as an input a vector $\mathbf{y} = \mathbf{c} + \mathbf{e}$, with $\mathbf{c}$ a codeword in C_B and $\mathbf{e}$ a $[t_1,t_2; v_1,v_2]_2^m$-bit error vector. The output here is an estimate $\mathbf{e}'$ of the error vector $\mathbf{e}$ (note that we use a slightly abnormal convention where the output is the error estimate rather than an estimate of the transmitted codeword. The codeword estimate can be computed as $\mathbf{c}' = \mathbf{y} - \mathbf{e}'$). Then, the decoder D_B operates in the following way. Each of the other D_i are the decoders for the corresponding codes C_i.

1. Form the vectors $(\mathbf{s}_1^0,\ldots,\mathbf{s}_n^0)$ from the decoder $D_2(H_2 \cdot (H_1' \cdot \mathbf{y}_1^T,\ldots, H_1' \cdot \mathbf{y}_n^T)^T)$.
2. Set the error $\mathbf{e}^*$ to be $(D_1'(\mathbf{s}_1^0),\ldots,D_1'(\mathbf{s}_n^0))$.
3. Set the codeword $\mathbf{y}'$ to be $\mathbf{y} + \mathbf{e}^*$.
4. Compute $(\mathbf{s}_1',\ldots,\mathbf{s}_n')$ from $D_2(H_2 \cdot (H_1' \cdot \mathbf{y}_1'^T,\ldots,H_1' \cdot \mathbf{y}_n'^T)^T)$.
5. Set $(\mathbf{s}_1'',\ldots,\mathbf{s}_n'')$ to be $D_3(H_3 \cdot (H_1'' \cdot \mathbf{y}_1'^T,\ldots,H_1'' \cdot \mathbf{y}_n'^T)^T)$.
6. Set I to be $\{i : (\mathbf{s}_i',\mathbf{s}_i'') \neq (\mathbf{0},\mathbf{0})\}$.
7. Let $\mathbf{y}''$ satisfy $\mathbf{y}_i'' = \mathbf{y}_i$ if i is in I and $\mathbf{y}_i'' = \mathbf{y}_i'$ if i is not in I.
8. Set $(\mathbf{s}_1^1,\ldots,\mathbf{s}_n^1)$ to be $D_3(H_3 \cdot (H_1'' \cdot \mathbf{y}_1''^T, \ldots, H_1'' \cdot \mathbf{y}_n''^T)^T)$.
9. $\mathbf{e} = (\mathbf{e}_1,\ldots,\mathbf{e}_n)$ where $\mathbf{e}_i = \mathbf{e}_i^*$ if i is not in I and otherwise $\mathbf{e}_i = D_1(\mathbf{s}_i^0,\mathbf{s}_i^1)$.

The basic idea here is to first correct the errors with fewer bits in error. Of course, some errors have too many bits in error, so they will be miscorrected (but only to at most weight $v_1 + v_2$). Next, we detect which errors are the miscorrected ones, and correct them as well.

We also note that all of the non-trivial operations in the decoding procedure are uses of the decoding functions D_1, D_2, D_3. Moreover, each of these operations is performed at most twice. Therefore, the overall decoding complexity is a small constant factor times the complexity of the worst (in terms of complexity) constituent code. So, for example, if we use BCH codes as our constituent codes, our overall decoding algorithm has complexity roughly twice as large as the largest constituent BCH code.

As mentioned, we can test the proposed graded-bit error-correcting codes on actual data collected from TLC flash devices. The data was collected in the following way: random data patterns are written to the device, filling each block. This procedure is repeated for 5000 program/erase (P/E) cycles; each 100 cycles, the data is read back for errors [1].

The comparisons were performed against other BCH codes with the same rate and length. In the three plots in Fig. 11.3, codes had lengths of 4096, 8192, and 16,384, respectively. The purple curve in the bottom figure, for example, indicates a graded bit error correcting code with parameters $[t_1,t_2; v_1,v_2] = [242,8; 1,3]$. The red curves represent 8-ary BCH codes. The blue curves represent (identical) binary BCH codes used to protect each of the 3 bits in the cell separately. The black curves

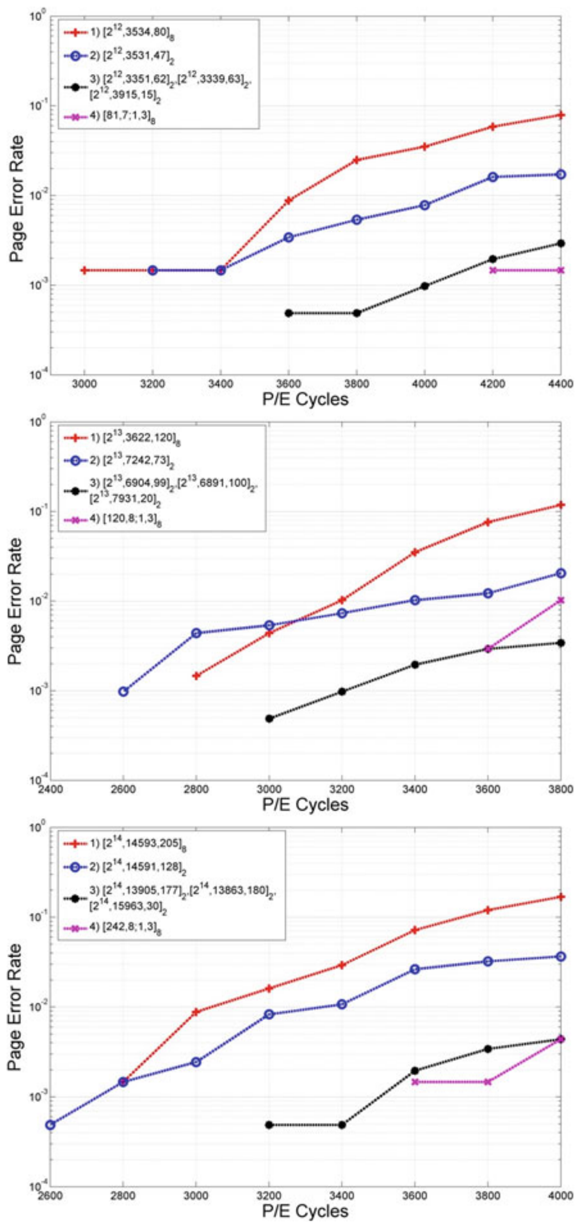

Fig. 11.3 Page error rates (PERs) for codes of with approximately the same length (4096, 8192, and 16,384 bits, respectively) and rate tested using data collected from TLC flash devices after varying numbers of program/erase cycles. The *red*, *blue*, and *black curves* use BCH codes (non-binary, identical binary over the separate pages, and differing binary codes over the separate pages, respectively.) The *purple curves* show our graded-bit error-correcting code construction. As can be seen, our asymmetric construction results in no errors (perfect operation) until much later in the device lifetime compared to traditional codes

represent three binary BCH codes (with different parameters) selected to optimize the error rate on each bit separately. For our graded bit error-correcting code constructions, we also selected BCH codes as our underlying constituent codes, so that the final parity check matrix H_B is produced by stacking the tensor products of parity check matrices of BCH codes.

As can be seen, using graded bit error-correcting codes allows for no errors at all on the measured data until late in the device lifetime (past 3000 P/E cycles). After this point, these codes perform as well or better than the separate binary BCH codes. Meanwhile, the 8-ary symmetric scheme has by far the worst performance.

We must point out that this is not the limit of what can be accomplished by the use of asymmetric codes tailored to handle specific error patterns. We have observed several other types of errors as well [4]. In the case of TLC flash, a careful study of the error patterns indicates that cells can be broadly divided into reliable and unreliable cells, where the unreliable cells are vastly more likely to be in error. In the case of the data set we examined, we noted that a specific set of roughly 65,000 cells (forming approximately 0.05 % of the total number of cells) resulted in more than 50 errors each across the 5000 P/E cycles from the test. In other words, about 10^{-4} of the cells account for more than 10 % of the errors.

What is the behavior of these unreliable cells? We observed that the cells produce these errors specifically when they are programmed to a higher voltage level. TLC cells have 8 possible voltage levels; frequent errors occurred when the unreliable cells were programmed to levels 4–7 (but not levels 0–3). Therefore, it is desirable to introduce a code that has the same features as the graded-bit-error-correcting codes previously discussed while also avoiding programming the unreliable cells to dangerously high levels.

Fortunately, this turns out to be possible. We restrict ourselves to the specific case of TLC flash, which means that we wish to create a code in GF(8), or, equivalently, a binary code of length 3n. Of course, a similar construction can be created for more general cases as well.

Before we proceed, let us introduce an auxiliary code construction, known as "stuck-at" error-correcting codes. First, let the operation $\circ$: $GF(2)^m \times GF(3)^m$ to $GF(2)^m$ be defined so that $\mathbf{b} = \mathbf{a} \circ \mathbf{s}$, where $b_i = s_i$ if $s_i < 2$ and $b_i = a_i$, otherwise. P_j is defined as the set of all vectors $\mathbf{s} = (s_1, \ldots, s_m)$ in P_j so that $|\{i : s_i < 2\}| \leq j$. Then, stuck-at error-correcting codes are defined in the following way:

Definition 5 For positive integers m,k,t,j, a $[m,k,t,j]_2$ binary code *C* is a linear code of length m and dimension k over GF(2) with encoding and decoding maps E_C and D_C such that

1. For all s in P_j and any message h, $E_C(h,\mathbf{s}) \circ \mathbf{s} = E_C(h,\mathbf{s})$, and
2. For any error vector $\mathbf{e}$ in $GF(2)^m$ with $wt(\mathbf{e}) \leq t$, $D_C(E_C(h,\mathbf{s}) + \mathbf{e}) = h$.

The idea behind Definition 5 is that even if a particular subset of cells is stuck (the subset has size at most j), the stuck-at error-correcting code *C* can still recover from errors. As we will see, we can use these types of codes as a building block for our construction; we adapt the stuck-at-error behavior to limit the levels of our target cells.

Next, we introduce our goal codes:

Definition 6 Let n,k,t_1,t_2,j be positive integers where $j,t_1,t_2 < n$. Then, a $[3n,k,t_1,t_2, j]$ dynamic bit-error-correcting code *C* is a binary linear code of length 3n and dimension k that is capable of correcting any $[t_1,t_2]$-bit-error vector. There is an additional constraint: if we write a codeword in *C* as $\mathbf{c} = (c_1, c_2, \ldots, c_n)$, where each c_i is an element in GF(8), then, given a set I of size at most j, $c_i \leq 3$ for all i in I and all codewords **c** in *C*.

Definition 6 matches Definition 4 from our previous discussion, but adds the requirement that a particular subset of cells is programmed to low levels only. Therefore, we introduce a construction that builds on Construction 2 and adds a corresponding constraint. Let us also use a simple map from elements in GF(4) to binary vectors of length 2 (again, α is a primitive element in GF(4)):

$$\Gamma(\alpha) = (0{,}1)^T, \quad \Gamma(\alpha^2) = (1{,}1)^T, \quad \Gamma(\alpha^3) = (1{,}0)^T, \quad \text{and} \quad \Gamma(0) = (0{,}0)^T.$$

Construction 3 Let $H_1 = (\alpha\ \alpha^2\ \alpha^3)$ and $H_2 = (1\ 1\ 1)$, where H_1 is a matrix in $GF(4)^{1\times 3}$ and H_2 is a matrix in $GF(2)^{1\times 3}$. Let H_3 be a parity check matrix for a $[n,k_3,t_1+t_2]_4$ code C_3. Also, let H_4 be a parity check matrix for a $[n,k_4,t_2,j]_2$ stuck-at-error correcting code (as introduced in Definition 5). Then, a parity check matrix for a $[3n,2k_3+k_4,t_1,t_2,j]_2$ dynamic bit-error-correcting code of length 3n is given by

$$H = \begin{bmatrix} \Gamma(H_3 \otimes H_1) \\ H_4 \otimes H_2 \end{bmatrix}.$$

The basic idea here is to slightly modify our previous graded-bit error-correcting construction (Construction 2) by forcing the use of a stuck-at error-correcting code construction. Rather than use it to specifically correct stuck-at errors, we do almost the reverse. The construction allows for mapping a message to one of several possible codewords, in order to deal with the stuck-at behavior. We take advantage of this by selecting the codeword where our unreliable cells are at lower levels.

For the case of our dynamic bit-error correcting codes, too, we can perform simulations to show the advantages of such codes. We perform the comparison against graded bit-error correcting codes (which lack the specific constraint of avoiding high levels in unreliable cells) and other previously mentioned codes, including various BCH codes.

In Fig. 11.4, the page error rates (PERs) are shown for codes of lengths 4096, 8192, and 16,384, respectively. As before, the lengths and rates of the codes compared against are the approximately equal. The same code constructions are shown as before; the green curve shows the new dynamic-bit-error code construction. Exactly 2 unreliable cells were forced to lower levels in the top two plots, while 4 unreliable cells were used in bottom plot. The purple curve shows the graded-bit error-correcting codes. The other curves are the same types of codes used for comparison in the previous sections.

Note that the dynamic-bit error-correcting codes have the best overall PER, while still not exhibiting any errors until very far in the lifetime of the device. The additional asymmetry of these codes (compared to the graded-bit error-correcting codes) has granted us an additional half an order of magnitude in PER performance. Therefore, we have the best of both worlds: codes with very good PERs, which do

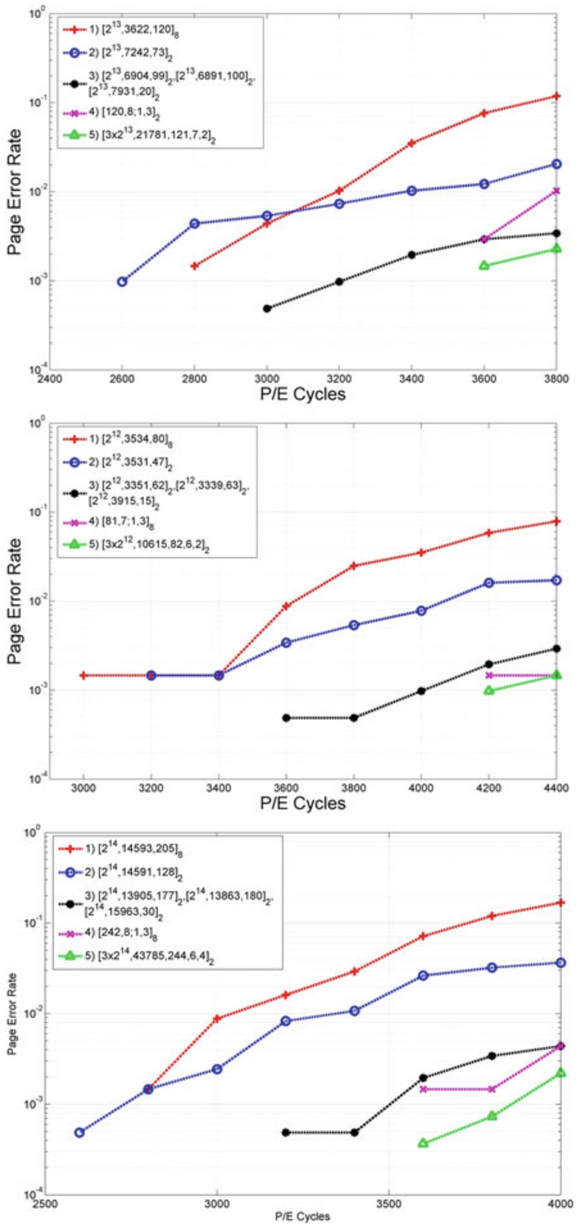

Fig. 11.4 Page error rates (PERs) for codes of with approximately the same length and rate tested with data collected from TLC flash devices after varying numbers of program/erase cycles. As before, the *red*, *blue*, and *black lines* show varying BCH-based code constructions. The *purple curve* shows the graded bit-error correcting code construction. The *green curves* are the new dynamic-bit error-correcting codes, which continue to show no errors until late in the device lifetime, but also offer an additional half-magnitude improvement in overall PER over the graded construction

not allow for any errors at all until late in the operative lifetime of the flash devices. We have successfully taken advantage of asymmetry to produce codes which are a dramatic improvement over traditional, symmetric codes.

11.1.2 Dynamic Thresholds

Another particularly interesting feature of the flash channel is the fact that it is time-varying. The longer the period of time between data write and data access operations, the higher the probability of read error. This property is due to certain physical effects acting on flash transistors. For example, over time, the electrons trapped on the floating gates of flash cells will leak out, escaping these gates. The errors caused by such effects are therefore inherently asymmetric.

In addition to these asymmetries, the time-varying character of the channel is also not considered or exploited by traditional coding techniques. Recall that flash devices work in the following way: the amount of charge on the floating gate is measured and compared to a set of thresholds. The result of this comparison determines the discrete value read out from the device. The thresholds used are traditionally fixed and permanent, ignoring the degradation of the channel over time. Although these fixed thresholds may be suitable for the channel at a particular period of operation, they often prove to be inefficient for differing retention periods.

One solution to this problem is to introduce *dynamic* thresholds, which can be changed over time. Although there are many ways to accomplish this task, there is a particularly simple approach. We set the thresholds in such a way that the distribution of the values in a block of cells is identical when being read as it was upon write [5, 6]. In other words, we use this distribution of values as side information.

Let us formalize this idea. Say that we have a block $\mathbf{x} = (x_1,x_2,\ldots, x_n)$ of n cells that can take on any of the q values $(0,1,\ldots, q - 1)$ each. In TLC flash, as in our previous discussion, $q = 8$. Now, some time passes and the written values have become the real values $\mathbf{v} = (v_1,v_2,\ldots, v_n)$. We have the thresholds $\mathbf{t} = (t_1,\ldots, t_{q-1})$, which we use to read $\mathbf{v}$ in the following way: the output $\mathbf{y} = \mathbf{t}(\mathbf{v})$ is given by

$$y_i = a, \quad \text{if} \quad t_a \le v_i \le t_{a+1},$$

where we take t_0 to be negative infinity and t_q to be positive infinity.

Now, let us denote by $\mathbf{k} = (k_0,\ldots,k_{q-1})$ the distribution of values in $\mathbf{x}$. That is, $k_a = |\{i \mid x_i = a, \ 1 \le i \le n\}|$. Thus, for example, $\mathbf{x} = (1,0,0,3,1,1,1,2)$ has $\mathbf{k}(\mathbf{x}) = (2,4,1,1)$, since $\mathbf{x}$ has 2 values of 0, 4 values of 1, and so on.

Then, we can define dynamic thresholds in the following way:

Definition 7 A threshold vector **t** is a *dynamic* threshold if

$$\mathbf{k}(\mathbf{y}) = \mathbf{k}(\mathbf{t}(\mathbf{v})) = \mathbf{k}(\mathbf{x}).$$

For example, say that the vector **x** above was written, and the real charge values are given by

$$\mathbf{v} = (1.2,0.2,0.6,2.3,1.1,1.0,1.3,2.2).$$

Then, if we use the fixed threshold $t^1 = (0.5,1.5,2.5)$, we would read the output

$$\mathbf{y}^1 = \mathbf{t}^1(\mathbf{v}) = (1,0,1,2,1,1,1,2),$$

with errors in the third and fourth positions. However, t^1 is not a dynamic threshold: the distribution of y^1 is $k(y^1) = (1,5,2,0)$, which is not equal to $k(x) = (2,4,1,1)$. Let us instead select a dynamic threshold $t^d = (0.7,2,2.25)$. Then, we correctly read

$$\mathbf{y}^d = \mathbf{t}^d(\mathbf{v}) = (1,0,0,3,1,1,1,2).$$

Of course, dynamic thresholds do not guarantee that the read sequence is error-free. However, they reduce error rates, since to yield an error, two components (with differing initial values) must have their values switched relative to each other. For example, if we have $x_i < x_j$, we must have $v_i > v_j$ to cause an error. This event occurs with lower probability in comparison to simply requiring $x_i < t_{xi}$, which is sufficient for an error in the fixed threshold case.

An illustration of this claim is in Fig. 11.5, where we simulated the degradation of the channel with the passage of time by modeling flash cells as Gaussians with increasing standard deviation over time. We then simulated a block of 10^5 cells by writing random values and reading back for errors using dynamic thresholds versus fixed thresholds. As the standard deviation of the Gaussians modeling the flash channel increases, the dynamic threshold scheme yields a much slower growth in error probability.

This type of simulation offers experimental support to the suggestion that dynamic thresholds outperform fixed thresholds. However, we add a theoretical comparison as well. Let us say that that $N(\mathbf{x},\mathbf{y})$ is the Hamming distance between vectors $\mathbf{x}$ and $\mathbf{y}$. If $\mathbf{y}$ is generated by reading $\mathbf{x}$, that is, $\mathbf{y} = \mathbf{t}(\mathbf{v}(\mathbf{x}))$ is the value read from $\mathbf{v}$ (itself formed by the written values $\mathbf{x}$) using the threshold $\mathbf{t}$, then, we write $N(\mathbf{x},\mathbf{y})$ as $N(\mathbf{t})$. Let us say also that $\mathbf{t}^*$ is the optimal threshold in the sense that $\mathbf{t}^* = \min_t N(\mathbf{x},\mathbf{y})$ for some fixed $\mathbf{x},\mathbf{y}$.

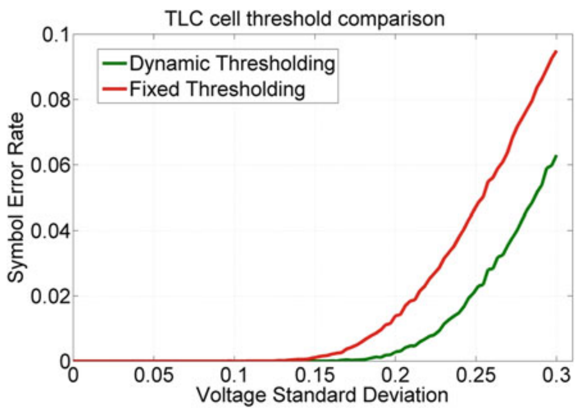

Fig. 11.5 Bit error rates (BERs) in a simple experiment modeling multi-level flash cells as Gaussians with increasing standard deviations over time. Blocks contain 10^5 cells. Dynamic thresholds and fixed thresholds were compared; as can be seen, dynamic thresholds offer improved performance versus traditional fixed thresholds

We also make the assumption that the maximum possible error magnitude is given by r, for some r in {0,…, q − 1}. This is a reasonable assumption for flash: we expect most errors to be of small magnitude, possibly at most 1. With this, we can say that any dynamic threshold $\mathbf{t}^{d}$ is quite close to the optimal threshold $\mathbf{t}^*$:

$$N(\mathbf{t}^{d}) \Leftarrow (r+1)N(\mathbf{t}^{*}).$$

In other words, any dynamic threshold is at most a constant factor (depending on the maximum error magnitude) from the optimal threshold. Of course, this optimal threshold requires knowledge of **x** itself to compute. This knowledge is not available when reading: a reliable estimate of **x** is the goal of the read operation. In other words, the dynamic threshold offers a practical solution that is quite close to an unobtainable optimum.

So far we have not discussed how to generate a set of thresholds. This is, of course, an important practical concern. There are two possible approaches (and a variety of combinations of the two) available [6]. The first is to use the distributions of values in blocks as side information.

This side information is then stored elsewhere. For example, we can store these values in very robust, highly-reliable cells, protected by powerful codes, which can then be read with fixed thresholds with low risk of error.

Another approach is to store data in constant-weight codewords. These codewords have a fixed distribution of values. Since the distribution is fixed, it can be hardcoded into the system from production, bypassing the need to communicate side information at all. The tradeoff here, of course, is the fact that constant-weight codes eliminate certain codewords from being used, yielding a potentially smaller overall rate.

With either approach, we must further protect our system with error-correcting codes. Dynamic thresholds by themselves will not sufficiently reduce the system's error rate to the target rate. This leaves us with the question of what choice of code to select. We could, of course, use an existing, off-the-shelf code, such as BCH codes. However, these schemes ignore the fact that dynamic thresholds yield asymmetric errors, in the same way that asymmetry in 3-bit TLC error vectors are ignored (leading to our improved tensor product-based constructions.) For example, a single component error in a vector cannot occur, since this would change the read codeword distribution, which is not possible with dynamic thresholds by definition. However, traditional codes cannot take advantage of this idea.

Instead, we can propose specialized asymmetric codes that operate specifically on dynamic thresholds.

Definition 8 Let a vector **x** be stored in a system with dynamic thresholds. An error **e** that **x** can experience under dynamic thresholding is called a [t,v]-**DT error** if **e** has at most t non-zero components and if each component has magnitude at most v. A code capable of correcting any [t,v]-DT error is called a [t,v]-**dynamic threshold error correcting (DTEC) code**.

Note that not all [t,v]-error vectors are [t,v]-DT error vectors. For example (1,0,0,0) is a [1,1]-error vector of length 1, but not a DT error vector, since with dynamic thresholds, errors require at least two positions to be non-zero in order to preserve the distribution of values between **x** and **x** + **e**. In other words, there are fewer DT-error vectors than there are error vectors in general. Although a conventional error-correcting code can correct DT errors, it also corrects error vectors which cannot occur with DT errors, which reduces the overall performance of the code by sacrificing rate for unused error-correction strength.

We introduce a type of asymmetric construction that specifically corrects 2 DT errors of any magnitude, thus providing an example of a [2,q − 1]-DTEC code construction:

Construction 4 Let *C* be a $[n,n-2]_q$ linear block code (of length n and dimension n − 2) over a field F_q with parity-check matrix given by

$$H = \begin{bmatrix} a_1 & a_2 & \dots & a_n \\ a_1^2 & a_2^2 & \dots & a_n^2 \end{bmatrix},$$

where $S = \{a_1, a_2, \ldots, a_n\}$ is a subset of distinct elements of F_q. Then, *C* is a [2, q − 1]-DTEC code if S is a Sidon set (a set with the property that for any four distinct elements a,b,c,d in S, $a + b \neq c + d$).

Note that such a code corrects any 2 errors in dynamic thresholding, while a general 2-error correcting code requires a much larger redundancy. This is the advantage of custom, asymmetric error-correcting codes. It is possible to modify the previous construction to yield codes for other values of limited magnitude r smaller than q − 1.

With, this, we have seen a further example of how to take advantage of asymmetries in order to introduce superior algebraic error-correcting codes.

11.2 Non-binary LDPC Codes

Next, we switch our focus from algebraic codes to graph-based codes. Graph-based codes have a nice advantage over algebraic codes, since it is possible to decode using soft information. In other words, graph-based code decoders can take as inputs fractional (rather than integer) values. Algebraic codes, however, lack this ability. The use of soft information is particularly important for storage devices such as flash, since we can perform multiple reads of the data in order to retrieve more accurate decoder inputs. Soft information thus yields excellent error-correcting performance. We provide more detail on this concept later on in this chapter.

We are particularly interested in one of the most important class of graph-based codes, non-binary low-density parity-check (NB-LDPC) codes. LDPC codes were first introduced in Gallager's seminal doctoral thesis in the 1960s and rediscovered during the 90s. Binary LDPC codes have been extensively studied and have found use in numerous applications.

Non-binary LDPC codes, however, have remained somewhat less well-understood. An early work by Davey and MacKay [7] showed that non-binary

LDPC codes offer better performance compared to their binary counterparts. This performance scales up with the field size parameter. However, this performance gain comes at the cost of decoder complexity. The initial implantation of the LDPC belief-propagation decoder for non-binary codes had a complexity of $O(q^2)$ for a field size of q. However, this complexity can be reduced to a more manageable $O(q \log q)$ by an FFT-based decoder implementation. Other techniques for low-complexity decoding have been proposed, including ones based on linear programming.

In addition to improved decoder complexity, a large number of constructions for non-binary LDPC codes have been proposed over the last ten years. The approaches taken for such constructions vary widely; for example, constructions include quasi-cyclic codes (some based on geometric approaches), protograph-based codes, quantum LDPC codes, and many others [8–10]. The proliferation of such improved works in the non-binary LDPC area of study suggests that such codes are approaching common, practical application. In general, increasing the code length of an LDPC code improves its performance; however, there are diminishing returns. For example, doubling the code length from 1000 bits to 2000 bits typically has a much greater positive effect on performance than doubling the code length from 100,000 bits to 200,000 bits.

However, before common application of non-binary LDPC codes can become reality, there is an additional roadblock to handle. This is the so-called LDPC “error floor”. The terminology reflects the appearance of the bit-error or frame-error rate versus SNR for LDPC codes. Initially, as the SNR increases, the BERs/FERs correspondingly improve dramatically; this is the “waterfall regime.” However, after a certain point, these curves become increasingly flat, entering the error-floor region. This error floor is a particularly important problem, since many applications for LDPC codes, such as data storage devices, operate at very high SNRs. For example, for Flash memory, the desired output FER often exceeds 10^{-15}; this point lies squarely in the error floor region for many LDPC codes. An illustration of an error floor is shown in Fig. 11.6.

What causes the error floor behavior? This is an important question that has been closely studied in the context of binary LDPC codes [11, 12]. We thus focus our attention on higher performance, non-binary LDPC codes. We begin by explaining the operation of practical LDPC decoders. In the case of storage devices, for example, a small number of probes of the underlying device are allowed. If there is only such probe permitted, we refer to the system as hard-decision. With more than one read, the system is soft-decision, as shown in the bottom of Fig. 11.7. However, only a small number of probes are allowed, due to latency issues. Note that an important problem is setting the reference thresholds (V_{R1},V_{R2},V_{R3} for the single-read case and V_{R1},…,V_{R6} for the two-read case.) A method based on mutual-information optimization was presented in [13].

As a result of the small number of reads, the continuous channel of the storage device has been transformed into a discrete channel. Similarly, in a digital system, the messages are quantized to finite-precision variables. As a result, in practical systems, decoder behavior ends up resembling that of decoders operating over

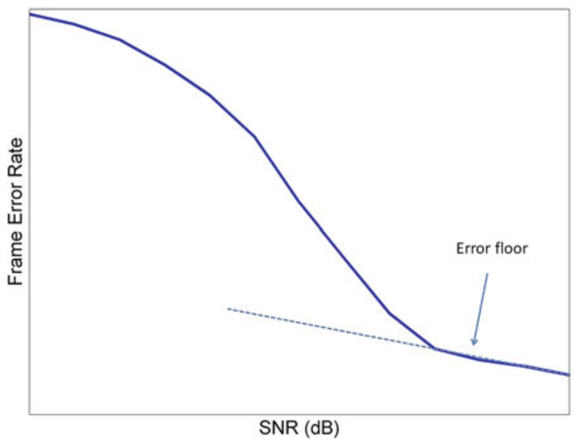

Fig. 11.6 Illustration of the "error floor" behavior of LDPC codes. Initially, as the SNR increases, there is a sharp downward slope as the frame error rate (FER) decreases. However, this slope eventually levels off, leading to much smaller improvement in FER for high SNRs

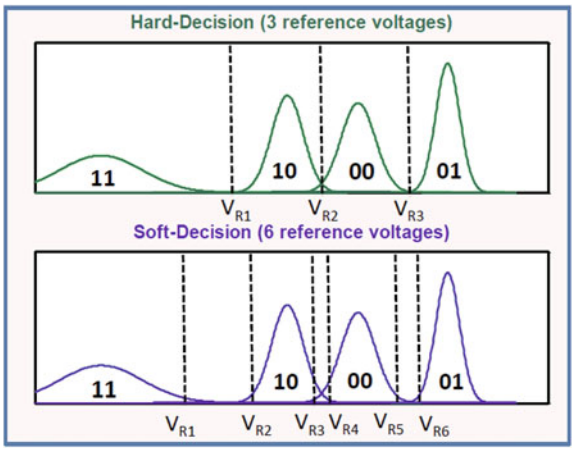

Fig. 11.7 Example of reads in a MLC (4-level) flash device. Hard decision allows only one read, and thus only one output state. For this reason, there is a single distinct threshold separating each state. Soft decision allows for multiple reads; 2 reads are shown on the *bottom figure*. There are many strategies for where to place the thresholds V_{Ri}

much simpler channels, such as discrete memoryless channels. LDPC codes over such channels are very well studied.

The majority of this existing research examines the error floors of binary codes. The error floor is in fact intimately connected to certain objects in the graph structure of the LDPC code. This graph structure is the Tanner graph; the Tanner graph of an LDPC code is a bipartite graph where the two classes of nodes are variable nodes (corresponding to components in LDPC codeword vectors) and check nodes (corresponding to the parity-check equations.) There is an edge between a check node and a variable node if the corresponding component is involved in the corresponding check equation, respectively. An example of a Tanner graph for a Hamming [7,4] binary code is shown below (Fig. 11.8). Of course, this code is not low-density; however, the simple parity-check matrix helps illustrate the idea behind the definition of the Tanner graph.

Since belief-propagation decoders operate on this graph structure, it is not surprising that certain configurations of nodes cause decoding problems. Trapping sets

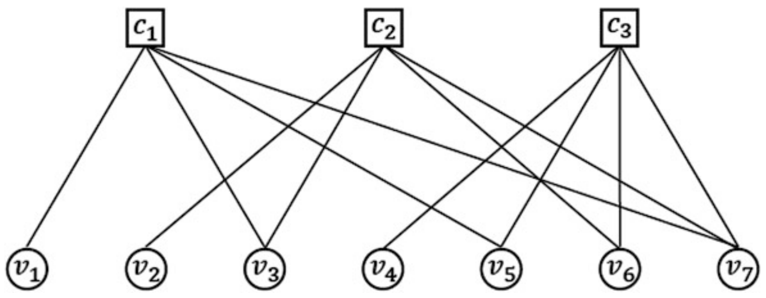

Fig. 11.8 Example of a Tanner graph for the (non-LDPC) Hamming [4, 7] binary code. The *circles* represent the 7 variable nodes corresponding to each bit in the codeword. The *squares* represent the three check equations from the code's parity-check matrix. There is an edge between a check node and a variable node if the corresponding bit is used in the parity-check equation

and absorbing sets are examples of subgraph objects that, when found in the Tanner graph of a particular code, are known to cause errors. These objects have been extensively studied in the case of binary LDPC codes. Many papers have proposed design algorithms for LDPC codes in order to avoid trapping and absorbing sets and thus to remove the error floor behavior [14, 15].

However, this problem is more challenging in the non-binary LDPC case. In this part of the chapter, we explore how to identify, enumerate, and remove absorbing sets for non-binary LDPC codes. We begin with a summary of absorbing sets in the traditional, binary LDPC case.

11.2.1 Binary Trapping/Absorbing Sets

We start with a subgraph of the Tanner graph for a binary LDPC code. The subgraph contains the variable node set V with |V| = a. The variable nodes in V are set to 1 while all other variable nodes are set to 0. The check nodes connected to the vertices in V are divided into sets E and O, where E contains check nodes with an even number of edges to vertices in V, and O has check nodes with an odd number of such edges. Of course, in this configuration, E contains satisfied check nodes while O contains unsatisfied check nodes. Now we can introduce trapping and absorbing sets:

Definition 9 V is an (a,b) trapping set if |O| = b.

Definition 10 V is an (a,b) absorbing set if |O| = b and if each variable node in V has (strictly) more neighbors in E than it does in O.

Definition 11 An elementary absorbing set/trapping set is an absorbing set/trapping set with the added condition that each of the neighboring satisfied check nodes has two edges connected to the set, while each of the neighboring unsatisfied check nodes has exactly one edge connected to the set.

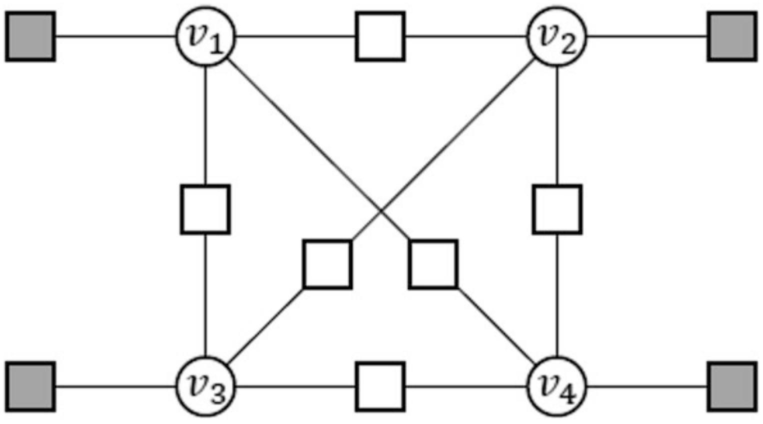

Fig. 11.9 Illustration of a (4,4) binary absorbing set. The *white circles* represent the four variable nodes in the absorbing set. The *white squares* are satisfied check nodes while the *gray squares* are unsatisfied check nodes. Since each of the unsatisfied check nodes has exactly one edge to the variable node set, this is an elementary (4,4) absorbing set

We show an illustration of such a set in Fig. 11.9.

We see that the configuration shown produces a (4,4) absorbing set. We have 4 variable nodes that are connected to 4 unsatisfied check nodes. The unsatisfied check nodes are gray squares, while the satisfied check nodes are white. Note that in addition, this is an elementary (4,4) absorbing set, since each of the unsatisfied check nodes has exactly one edge connecting it to the 4 variable nodes, while each of the satisfied check nodes has exactly two edges to the variable nodes.

Notice the basic idea of the absorbing set: it is a configuration of variable nodes where a majority-logic bit-flipping decoder will make an error (here, we assume that the all zero-codeword was sent) but will be unable to recover from this error. This behavior occurs precisely because of the fact that the majority of the neighboring check nodes are satisfied.

We will also require a few additional graph theory concepts. We can define a vector space on the set of all cycles of an undirected graph. For such a graph G with G = (V,E), the power set of E, 2^E, is a vector space when taking symmetric set difference as the addition operation, the identity function as the negation operation, and the empty set as the additive identity element. Then, the cycle space of the graph G is the subspace of 2^E which has the cycles of G as its elements. Now we apply basic principles from linear algebra:

Definition 12 A set of cycles F in G = (V,E) is the cycle span of G if it forms a basis for the cycle space. Cycles in a cycle span are called fundamental cycles.

We can also introduce a related graph structure, called a variable node (VN) graph. This graph is defined based on the bipartite graph of an elementary absorbing set. The variable node graph contains only variable nodes; these nodes are connected by an edge if they share a degree-two check node as a neighbor.

11.2.2 Non-binary Absorbing Sets

We are now ready to tackle the matter of non-binary absorbing sets. Since we are working in the non-binary regime, the Tanner graph of a code has weights placed on the edges connecting variable and check nodes. This weight is equal to the corresponding non-zero value in the non-binary parity-check matrix of the LDPC code. This adds an additional aspect to the graph structure: there is a *topological* structure (just as is the case with binary codes), but also we now have a weight structure. As a result, non-binary absorbing sets must also satisfy weight conditions.

As before, we seek a configuration where each variable node has more satisfied neighboring check nodes compared to unsatisfied nodes; however, for satisfied/unsatisfied part to be the case, we will require certain relationships between the weights. We show an example of such an absorbing set in Fig. 11.10.

Note that here the edge weights are taken to be nonzero elements from the code's finite field GF(q). This absorbing set has the same topological structure as the previous binary absorbing set example.

In the general case, however, in order for the degree-two check nodes to be satisfied, we need the following relationships to hold over GF(q). Note that the weights are labeled on the earlier diagram.

$$\mathrm{v}_1\,\mathrm{w}_1 = \mathrm{v}_2\,\mathrm{w}_2,\quad \mathrm{v}_2\,\mathrm{w}_3 = \mathrm{v}_4\,\mathrm{w}_4,\quad \mathrm{v}_4\,\mathrm{w}_5 = \mathrm{v}_3\,\mathrm{w}_6,$$
$$\mathrm{v}_3\,\mathrm{w}_7 = \mathrm{v}_1\,\mathrm{w}_8,\quad \mathrm{v}_2\,\mathrm{w}_{11} = \mathrm{v}_3\,\mathrm{w}_{12},\quad \mathrm{v}_1\,\mathrm{w}_9 = \mathrm{v}_4\,\mathrm{w}_{10}.$$

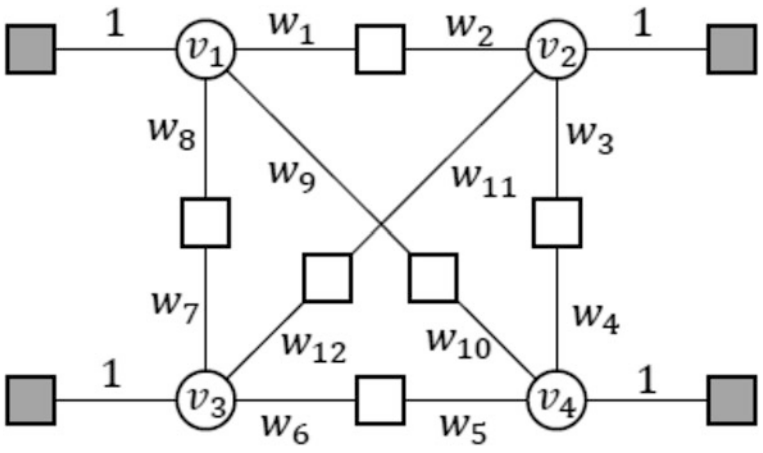

Fig. 11.10 Illustration of a non-binary (4,4) absorbing set. As before, this is an elementary absorbing set. Note that each edge now has a weight; these weights must satisfy certain conditions for the subgraph to form an absorbing set. However, the unlabeled version of the graph forms a binary absorbing set

Each of these equations comes directly from the definition of the Tanner graph. For example, the check node between v_1 and v_2 is only satisfied if (recall that all variable nodes except for $v_1,\ldots,v_4$ are set to 0, while $v_1,\ldots,v_4$ are set to 1) if the corresponding check equation is 0: $v_1w_1 + v_2w_2 = 0$.

If our field size is a power of 2, so that $q = 2^p$, we can eliminate the variable nodes in these equations in order to write a series of conditions exclusively for the weights:

$$w_1\, w_7\, w_{11} = w_2\, w_8\, w_{12}, \quad w_3\, w_5\, w_{12} = w_4\, w_6\, w_{11}, \quad w_2\, w_4\, w_9 = w_1\, w_3\, w_{10},$$

where, as before, the equations are taken over GF(2^p).

The basic idea can be written in a general form to define non-binary absorbing sets:

Definition 13 A set V is an (a,b) absorbing set over GF(q) if there exists an $(l - b)$ x a submatrix B of rank r_B given by elements $b_{j,i}$ for $1 \le j \le l - b$, $1 \le i \le a$ in matrix A satisfying the conditions:

1. If N(B) is the null space of B and $\mathbf{d}_i$, $1 \le i \le b$ is the *i*th row of D where D is given by excluding the matrix B from A, then, there exists $x = [x_1\ x_2\ \ldots\ x_a]^T$ in N(B) such that for x_i is non-zero for all i in {1,…,a} and there is no i such that $\mathbf{d}_i$ $x = 0$.
2. If D contains the elements $d_{j,i}$ for $1 \le j \le b$, $1 \le i \le a$, then, for all i in {1,2,… a}, then

$$\sum_{j=1}^{l-b} S\left(b_{j,i}\right) > \sum_{j=1}^{b} S\left(d_{j,i}\right)$$

Here the function S is an indicator function such that $S(x) = 1$ for x nonzero and 0 for $x = 0$.

We observe that a similar type of adaptation to the non-binary case is possible for trapping sets as well.

We can define non-binary elementary absorbing sets by adding the same condition as we did for the binary absorbing set to elementary binary absorbing set case. In this elementary absorbing set case, we can further manipulate the conditions above to have the following form, which resembles that as shown in our example. Let C_p be a cycle that contains p distinct variable nodes and p distinct check nodes in a graph induced by an (a,b) non-binary absorbing set. Let $C_p = c_1–v_1–c_2–v_2–\cdots–c_p–v_p–c_1$. The weight $w2_{i-1}$ is the label on the edge connecting c_i and v_i. Similarly, w_{2i} is the label on the edge connecting v_i and c_{i+1}. Then, we have the following.

Lemma 1 *If the field size parameter $q = 2^p$, then, every cycle C_p satisfies the following relationship:*

$$\prod_{k=1}^{p} w_{2k-1} = \prod_{k=1}^{p} w_{2k}.$$

In the case of elementary non-binary absorbing sets, we now have a simple breakdown of the definition: the (unweighted) topological structure must be a binary absorbing set, and, in addition, the weights must satisfy the equation given in Lemma 1.

Now that we have set up our definitions and identified just what a non-binary absorbing set is, we are ready to examine how to improve the performance of our non-binary codes.

11.2.3 Performance Analysis and Implications

We begin by identifying how frequently the weight conditions can be satisfied. This is an important question, since if the conditions are not met, we do not have an absorbing set. This concept is described in the following theorem.

Theorem 1 *We have an (a,b) unlabeled (binary) elementary absorbing set with e satisfied check nodes. Then,*

1. *A fraction of* $(q-1)^{a-e-1}$ *of edge weight assignments (taken over GF(q)) produce non-binary elementary absorbing sets.*
2. *A fraction of* $e(q-1)^{a-e-1}(q-2)$ *of edge weight assignments (again taken over GF(q)) produce (a,b + 1) non-binary trapping sets.*

The proof of the theorem relies on simple counting arguments based on the graph-theoretic ideas already introduced.

The theorem implies that there is a larger number (by a factor of $e(q-2)$) non-binary trapping sets compared to non-binary absorbing sets, assuming that the code and its weights are generated randomly. In practice, however, simulation results show that error profiles do not involve any errors from trapping sets. On the other hand, error profiles do show errors that are a result of absorbing sets. What explains this behavior? The idea is that quantization used in decoding algorithms results in these belief propagations acting in a way similar to majority-logic bit flipping decoders. Such decoders do not struggle with the more general trapping set errors, but, as we see from the definition of absorbing sets, these decoders will produce errors when faced with absorbing sets.

For this reason, it is more desirable to find a way to remove or reduce absorbing sets from the Tanner graphs of non-binary LDPC codes. First, we note that we must target certain absorbing set parameters over others. In the error-floor regime, which is at high SNR, errors typically only include a small number of variable nodes. Therefore, we look at small absorbing sets. In fact, the performance of the LDPC decoder will be dominated by the smallest absorbing set, which is also typically an elementary absorbing set. The goal thus becomes to maximize the size of the smallest absorbing set.

We are now ready to introduce an algorithm that eliminates problematic absorbing sets from non-binary LDPC code Tanner graphs. As described, the key idea is to manipulate the edge weights in such a way that the subgraphs involved are no longer absorbing sets. The algorithm is given below. We use one additional term: an absorbing set A is a child of an absorbing set B if A is a subgraph of B. We call B a parent of A.

Algorithm 1:

```
1.  Input: The Tanner graph G; edge weights taken over GF(q).
2.  Find $U_j$, the set of all $(a_j,b_j)$ absorbing sets in the binary, unlabeled version of G (using
    the existing techniques in [16] for the general case, or, in specific cases, more
    sophisticated approaches, such as the technique in [14] for circulant-based codes,
    including quasi-cyclic codes).
3.  Select W, the set of non-binary absorbing sets to be removed.
4.  X is the set of non-binary absorbing sets that cannot be removed.
5.  Start with X empty.
6.  A is the set of absorbing sets in W which have been processed (either eliminated
    from G or placed into the list X).
7.  Start with A empty.
8.  For every edge j in T, $C_j$ is defined to be the set of re-weighted cycles which include
    edge j.
9.  For all j in T, start with $C_j$ empty.
10. Find $(a_j,b_j)$, the smallest-size non-binary absorbing set in W\A.
11. If this set is a child of another absorbing set already in A, go to 31.
12. for all u in $U_j$,
13.   Find $F_u$, a set of fundamental cycles of u.
14.   Let $E_u$ be the set of edges in u.
15.   For an edge k in $E_u$, set $M_k$ to be the set of cycles in $F_u$ including k.
16.   if (5) is satisfied for all the cycles in $F_u$, then
17.     Find an edge i in $E_u$ with edge weight $w_i$ minimizing $|C_i|$
18.     if there is exists nonzero $w_i'$ (not equal to $w_i$) so that all cycles including I do not
    satisfy (5), then
19.       Replace $w_i$ with $w_i'$
20.     else
21.       Set $E_u$ to be $E_u$\i
22.       if $E_u$ is empty then
23.         Set X to be X union $U_j$ and go to 31.
24.       else
25.         Go to 17.
26.       end if
27.     end if
28.     For every edge e in the cycles of $M_i$, update $C_e$ to $C_e$ union $M_e$.
29.   end if
30.   end for
31.   Add (a,b) absorbing set to the list A.
32.   If A is not equal to W, go to 9.
33.   If X is empty, all absorbing sets have been removed. Otherwise, it is not possible
    to remove the sets in X.
```

The basic concepts in Algorithm 1 follow. First, we select the elementary absorbing sets that we wish to eliminate. These sets must be determined according to the code parameters (for example, column weight, girth, etc.). Next, among this group of sets, we examine the smallest absorbing set. We look for binary versions of this set in the unlabeled (that is, the binary version of the) Tanner graph. If the fundamental cycles of these absorbing sets satisfy the formula in Lemma 1, we modify the weight of one of the edges to some other non-zero element in GF(q). We make this choice in such a way that the previously removed absorbing sets are not brought back. The process continues once the current absorbing set has been removed.

Let us see an error profile (giving errors due to various absorbing sets) for a particular choice of code. We show the effects of the previous algorithm on these errors (Table 11.2).

Each of the error types refers to an (a,b) absorbing set which causes the error. The algorithm removes all of these absorbing sets (up to the (8,2) set), which dramatically reduces the number of errors. Here, the code is a non-binary LDPC code over GF(4). The length was 2904 bits, the SNR was 5.1 dB, the code rate was 0.878, and the column weight was 4.

Next, in Fig. 11.11, we show a performance plot showing the effect of using the algorithm (which is labeled A-method). We also compare with several other algorithms which also attempt to resolve error floors through absorbing/trapping set modification. In particular, we compare against the approach presented in [17], which we refer to as the 'P-method'. This approach attempts to cancel all cycles of length l in the Tanner graph, where l is between the girth g and a l_{max} parameter. This cancellation has the effect of removing certain absorbing sets as well (in particular very small ones.) Another method we compare against, which we refer to as the 'N-method', was proposed in [18].

Here, the figure shows frame error rate (FER) versus SNR for the original codes and the three methods discussed. Note that the previously introduced algorithm produces the best overall improvement in the FER. The code length was approximately 2930 bits, the rate was 0.88, the column weight was 4, and the QSPA-FFT decoder was used for decoding.

In the case of non-binary quasi-cyclic (NB-QC LDPC) codes, which are a very practical class of non-binary LDPC codes, we have the results shown in Fig. 11.12. Here the length is approximately 1400, the rate approximately 0.81, the column weight 4, and, again, the QSPA-FFT decoder was used.

Table 11.2 Error profile for non-binary LDPC code over GF(4)

Error type	(4,4)	(5,0)	(5,2)	(6,2)	(6,4)	(6,6)	(7,4)	(8,2)	others
Original	35	7	9	11	17	21	8	10	10
After Alg.	0	0	0	0	0	0	0	0	9

Code length is 2904 bits and code rate is 0.878. The code has column weight 4 and the SNR is 5.1 dB. Shown are the error profiles due to various absorbing sets before and after the absorbing set removal algorithm

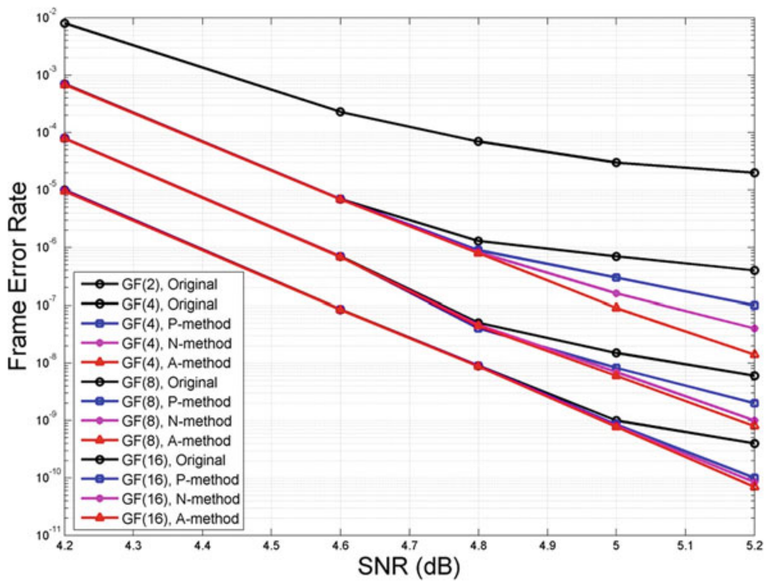

Fig. 11.11 SNR versus frame error rate (FER) for binary and non-binary LDPC codes over several fields. The codes had approximate length 2930 bits, rate 0.88, and column weight 4. The *curves* include the original, unmodified codes along with codes resulting from several methods aimed at improving non-binary LDPC codes. The method described in Algorithm 1 is labeled A-method; this method yields the most significant improvement in FER

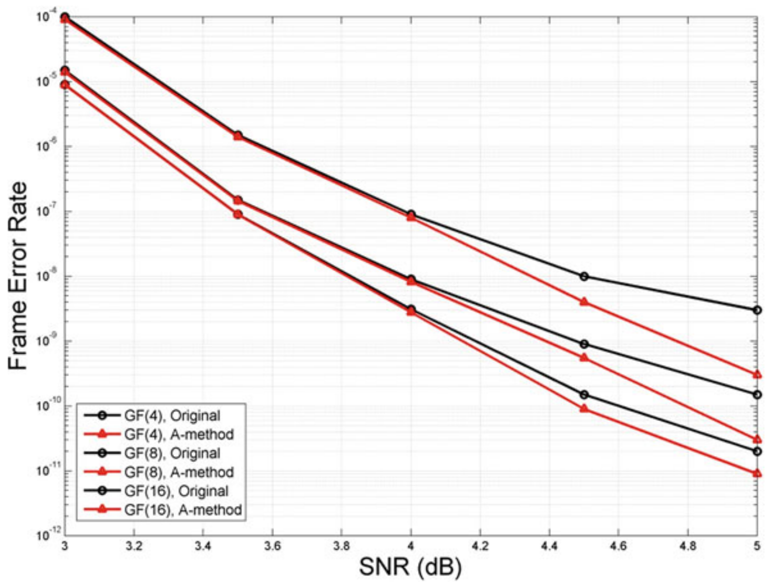

Fig. 11.12 SNR versus FER plot for non-binary QC-LPDC with different field sizes. We compare the original code to the codes improved by using the A-method from Algorithm 1. Note that the improvement is strongest in smaller field sizes

We note the fact that the improvement over the baseline diminishes as the field size q grows. The reason for this is the fact that since there are many more choices of edge weights for larger field sizes, absorbing sets naturally occur with smaller probability, so that there are fewer of them to remove through any of the possible algorithms.

11.3 Summary

In this chapter, two classes of non-standard codes were studied. The first class is composed of algebraic codes, which rely only on hard information and are suitable for applications where simple and efficient decoding is necessary, but error tolerance is more relaxed, such as inexpensive data storage devices. The second class is made up of LDPC codes, which have more complex decoding, but offer extremely good performance. LDPC codes are thus suitable for applications that require extreme reliability. Flash devices occupy the entire spectrum between these two endpoints. Both of the classes of advanced codes we study offer significant improvements over their traditional counterparts, but, at the same time, present more challenges in terms of constructions, design choices, and analyses.

First, we examined asymmetric algebraic codes. We motivated this study by looking at asymmetric channels modeling the physical channels of data storage devices. It was shown that using conventional symmetric codes was wasteful, either in terms of code rate or error-correcting ability. Two types of asymmetric codes were discussed: graded-bit error-correcting codes based on the tensor-product operation, and dynamic threshold-based codes relying on the dynamic thresholding side-information technique.

Afterwards, we looked at non-binary LDPC codes, which offer better performance compared to the frequently-studied binary LPDC codes. We examined the error-floor problem in the non-binary case and defined the underlying non-binary absorbing set objects that result in the error floor. We introduced an algorithm that can efficiently remove the problematic small absorbing sets from the Tanner graph of a non-binary LDPC code. Simulation results showed significant improvement over baseline non-binary LDPC codes.

References

1. E. Yaakobi, L. Grupp, P.H. Siegel, S. Swanson, J.K. Wolf, Characterization and error-correcting codes for TLC flash memories, in *Proceedings on IEEE International Conference on Computing, Networking, and Communications (CCNC)*, Maui, HI, Jan–Feb 2012, pp. 486–491
2. J.K. Wolf, An introduction to tensor product codes and applications to digital storage systems, in *Proceedings on IEEE Information Theory Workshop (ITW)*, Punta del Este, Uruguay, Oct 2006, pp. 6–10

3. R. Gabrys, E. Yaakobi, L. Dolecek, Graded bit-error-correcting codes with applications to flash memory. IEEE Trans. Inf. Theory **59**(4), 2315–2327 (2013)
4. R. Gabrys, F. Sala, L. Dolecek, Coding for unreliable flash memory cells. IEEE Commun. Lett. **18**(9), 1491–1494 (2014)
5. H. Zhou, A. Jiang, J. Bruck, Error-correcting schemes with dynamic thresholds in non-volatile memories, in *Proceedings on IEEE International Symposium Information Theory (ISIT)*, St. Petersburg, Russia, Jul–Aug 2011, pp. 2143–2147
6. F. Sala, R. Gabrys, L. Dolecek, Dynamic threshold schemes for multi-level non-volatile memories. IEEE Trans. Commun. **61**(7), 2624–2634 (2013)
7. M.C. Davey, D. MacKay, Low-density parity check codes over GF(q). IEEE Commun. Lett. **2** (6), 165–167 (1998)
8. A. Bazarsky, N. Presman, S. Litsyn, Design of non-binary quasicyclic LDPC codes by ACE optimization, in *Proceedings on IEEE Information Theory Workshop (ITW)*, Seville, Spain, Sep 2013, pp. 1–5
9. L. Dolecek, D. Divsalar, Y. Sun, B. Amiri, Non-binary protograph-based LDPC codes: enumerators, analysis, and designs. IEEE Trans. Inf. Theory **60**(7), 3913–3941 (2014)
10. I. Andriyanova, D. Maurice, J.-P. Tillich, Quantum LDPC codes obtained by non-binary constructions, in *Proceedings on IEEE International Symposium Information Theory (ISIT)*, Cambridge, MA, Jul 2012, pp. 343–347
11. L. Dolecek, Z. Zhang, V. Anantharam, M.J. Wainwright, B. Nikolic, Analysis of absorbing sets and fully absorbing sets of array-based LDPC codes. IEEE Trans. Inf. Theory **56**(1), 181–201 (2010)
12. T.J. Richardson, Error floors of LDPC codes, in *Proceedings on IEEE Allerton Conference on Communication, Control, and Computing*, Monticello, IL, Oct 2013, pp. 1426–1435
13. J. Wang, K. Vakilinia, T.-Y. Chen, T. Courtade, G. Dong, T. Zhang, H. Shankar, R. Wesel, Enhanced precision through multiple reads for LDPC decoding in Flash memories. IEEE J. Sel. Areas Commun. **32**(5), 880–891 (2014)
14. J. Wang, L. Dolecek, R. Wesel, The cycle consistency matrix approach to absorbing sets in separable circulant-based LDPC codes. IEEE Trans. Inf. Theory **59**(4), 2293–2314 (2013)
15. D.V. Nguyen, S.K. Chilappagari, M.W. Marcellin, B. Vasic, On the construction of structured LDPC codes free of small trapping sets. IEEE Trans. Inf. Theory **58**(4), 2280–2302 (2012)
16. M. Karimi, A.H. Banihashemi, Efficient algorithm for finding dominant trapping sets of LDPC codes. IEEE Trans. Inf. Theory **58**(11), 6942–6958 (2012)
17. C. Poulliat, M. Fossorier, D. Declercq, Design of regular $(2, d_c) -$ LDPC codes over GF(q) using their binary images. IEEE Trans. Commun. **56**(10), 1626–1635 (2008)
18. T. Nozaki, K. Kasai, K. Sakaniwa, Analysis of error floors of non-binary LDPC codes over MBIOS channel, in *Proceedings on IEEE International Conference on Communication (ICC)*, Kyoto, Japan, Jun 2011, pp. 1–5

Chapter 12
System-Level Considerations on Design of 3D NAND Flash Memories

Chao Sun and Ken Takeuchi

Abstract This chapter introduces the design of three-dimensional (3D) NAND flash memory with the implications from the system side. For conventional two-dimensional (2D) scaling, it is facing various limitations such as lithography cost and cell-to-cell coupling interference. To sustain the trend of bit-cost reduction beyond 10 nm technology node, 3D NAND flash memory is considered as the next generation technique. Further, emerging memories called storage-class memories (SCMs) such as resistive RAM (ReRAM), phase change RAM (PRAM) and magnetoresistive RAM (MRAM) will revolutionize the storage system design. By introducing SCM into the solid-state drive (SSD), hybrid SCM/3D-NAND flash SSD and all SCM SSD achieve much higher write performance than all 3D-NAND flash SSD due to SCM's fast speed. In addition, the performance of the SSD is workload dependent. Thus, it is meaningful to obtain the design guidelines of 3D NAND flash for both all 3D-NAND flash SSD and hybrid SCM/3D-NAND flash SSD with representative real-world workloads.

Keywords NAND flash memory · Storage-class memory (SCM) · Solid-state drive · Flash translation layer · Garbage collection

12.1 Introduction

There is a growing demand for NAND flash memory due to its fast speed, low power, and high reliability. NAND flash memory based storage systems are being widely used from consumer electronic products like SD cards to enterprise applications like solid-state drives (SSDs) in servers and data centers. SSDs designed for enterprise applications are discussed in this chapter. As described in Sect. 12.2, the

C. Sun (✉) · K. Takeuchi
Chuo University, Tokyo, Japan
e-mail: sun@takeuchi-lab.org

K. Takeuchi
e-mail: takeuchi@takeuchi-lab.org

bottleneck of NAND flash based SSDs lies in the write performance rather than the read performance due to the inherent characteristics of the NAND flash. Hence, the write performance of the SSD should be improved to meet the increasing requirement for high performance storage in this big data era.

On the other hand, storage class memories (SCMs) such as the resistive RAM (ReRAM), phase change RAM (PRAM) and magnetoresistive RAM (MRAM) are attracting more and more attention due to their faster speed, higher endurance and lower power consumption than the NAND flash memory. SCMs bridge the bandgap between the DRAM and NAND flash memory. According to the speed and capacity, SCM devices are divided into two categories: DRAM-like and NAND-like. DRAM-like SCMs are called memory-type SCM (M-SCM) such as the MRAM while NAND-like SCMs are named storage-type SCM (S-SCM) such as the ReRAM and PRAM. The hybrid M-SCM/3D NAND flash SSD and all S-SCM SSD have been proposed as the next generation SSDs.

In this chapter, techniques to improve the write performance of the SSD are introduced. Three SSDs including all 3D-NAND flash SSD, hybrid M-SCM/3D-NAND flash SSD and all S-SCM SSDs are discussed. Evaluated with representative real-world workloads, it is found that the write performance of the 3D-NAND flash-based SSDs is workload dependent. According to the system-level evaluation results, the design guidelines of the 3D-NAND flash for the SSDs are obtained.

12.2 Background of Solid-State Drive

Figure 12.1 shows the memory hierarchy of the computer system. Top layer's memories have a faster speed but smaller capacity (high bit cost). In contract, bottom layer's memories have a slower speed but larger capacity (low bit cost). SCMs, NAND flash and hard disk drive (HDD) are non-volatile. In the memory

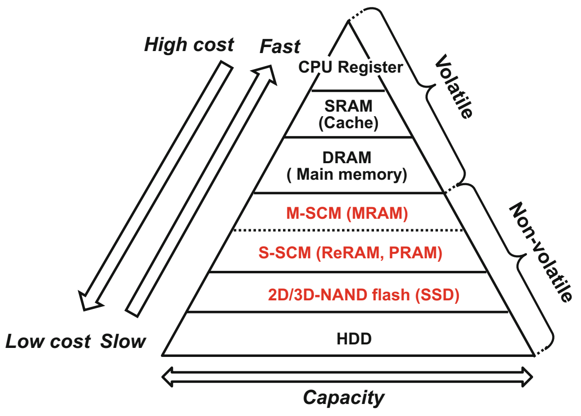

Fig. 12.1 Memory hierarchy

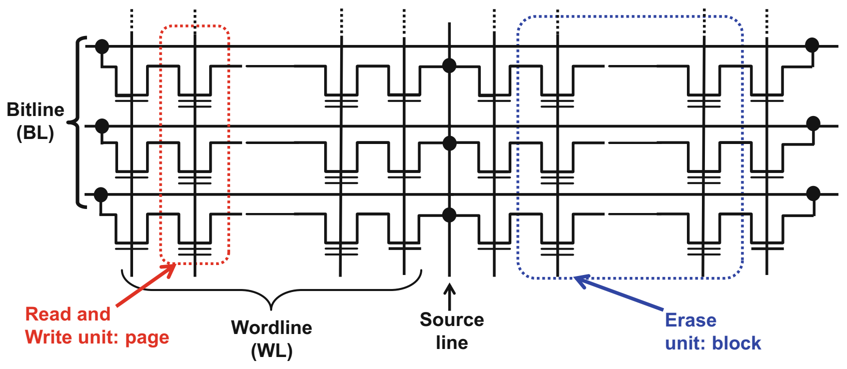

Fig. 12.2 NAND flash organization [1]

hierarchy, NAND flash memory lies between the SCM and HDD. Since the bit cost of the NAND flash memory is continuously reducing by scaling and multi-bit technology, SSD becomes cost-effective as an alternative of HDD.

Figure 12.2 illustrates the organization of the NAND flash memory [1]. The memory cells connected with the same word-line consists of a page, which is the read and write unit of the NAND flash. A block is the erase unit. There are typical 128–256 pages in a block for the multi-cell level (MLC) NAND flash.

The architecture of the all 3D-NAND flash SSD, hybrid M-SCM/3D-NAND flash SSD and all S-SCM SSD are described in Fig. 12.3. The key component of the SSD is the SSD controller that integrates the flash translation layer (FTL) enabling SSD to work as a block device. As shown in Fig. 12.4, the basic but very critical function in the FTL is the logical-to-physical address translation, required due to the prohibited in-place overwrite characteristics of the NAND flash memory. According to the mapping granularity, the address translation can be classified into the page-level mapping, block-level mapping and hybrid mapping. When a page data overwrite

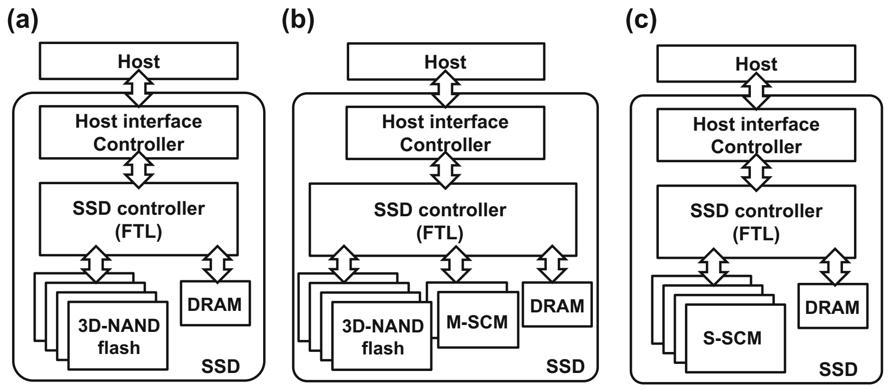

Fig. 12.3 SSD architectures of **a** all 3D-NAND flash SSD, **b** hybrid M-SCM/3D-NAND flash SSD and **c** all S-SCM SSD

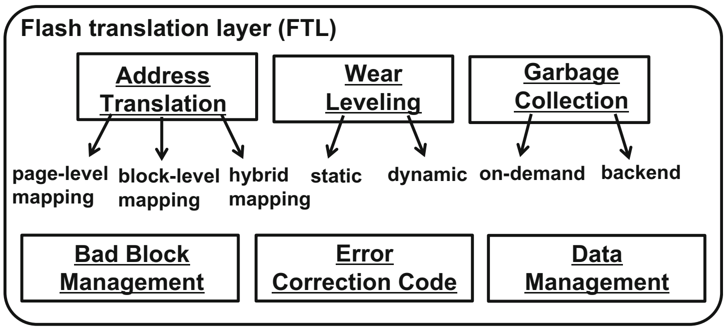

Fig. 12.4 Essential functions in the flash translation layer (FTL)

happens, the old data is read from the old page, merged with the new data, and written to a new page. After that, the old page is invalidated. Hence, there are three page statuses in the SSD: free page, page with the valid data and page with the invalid data. Frequently accessed data (hot data) will create massive number of invalid pages. When the SSD free space reduces to below a threshold (a few free blocks in a plane), an operation, garbage collection (GC), in FTL, is triggered on-demand or in the backend to reclaim one or more old blocks. Before erasing an old block, all the valid pages in the block have to be copied to the free spaces in another block, as shown in Fig. 12.5 [2]. Thus, the latency of the GC increases with the number of valid pages in the recycling block. When such page-copy overhead is large, the GC would become the bottleneck of the SSD write performance. Furthermore, the wear leveling in the FTL guarantees the NAND flash blocks are worn out evenly to maximize the lifetime of the SSD. According to whether the static data in NAND flash blocks are periodically moved around or not, wear leveling can be classified to static and dynamic wear leveling. Other functions like the error correction code (ECC) and bad block management (BBM) are also essential.

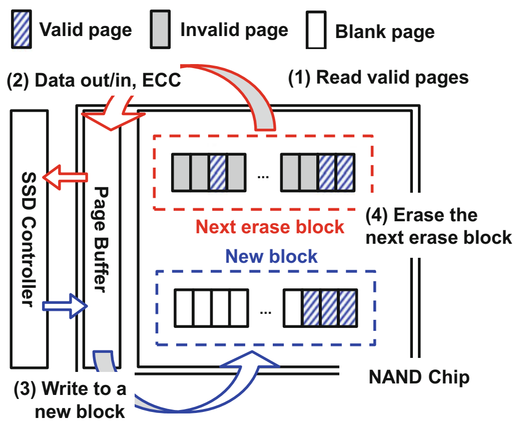

Fig. 12.5 Garbage collection (GC) operation [2]

12.3 SSD Performance Improvement Techniques

This chapter introduces three techniques to improve the write performance of 3D-NAND flash-based SSD: storage engine assisted SSD (SEA-SSD), logical block address (LBA) scrambled SSD and hybrid M-SCM/3D-NAND flash SSD. The first two techniques are based on the SSD controller and middleware co-design. The last technique introduces the M-SCM into the SSD system. Finally, the design of the all S-SCM SSD, as the long-term solution, is presented as well.

12.3.1 Storage Engine Assisted SSD (SEA-SSD)

Database is one of the most widely used applications in enterprise servers. The middleware, storage engine of the database, controls when and where the data should be stored in the storage. Therefore, the first technique, storage engine assisted SSD (SEA-SSD), co-designs the storage engine with the SSD controller to improve the SSD write performance for the database. It is based on the idea that the upper layer of the storage stack holds much richer information than the lower layer. For the current SSD, it only receives the information from the block device layer of the operation system (OS), which includes the data, data size and data address. The information is quite limited.

Figure 12.6 shows the comparisons between the conventional computer system and proposed computer system with SEA-SSD [3]. Due to reasons (i) the storage layers like the file system, block layer etc. are optimized for the conventional HDD but not for SSD, conventional OS is inefficient for SSD storage [4–8], (ii) great engineering effort is required if the hint messages have to go through all the layers in the OS, the OS is simply bypassed in the proposed SEA-SSD. Hints are passed from the SE (Storage Engine) to the SSD controller in order to store data more efficiently.

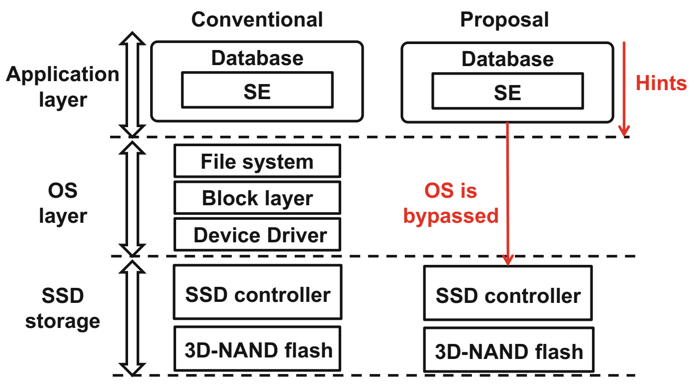

Fig. 12.6 SEA-SSD concept [3]

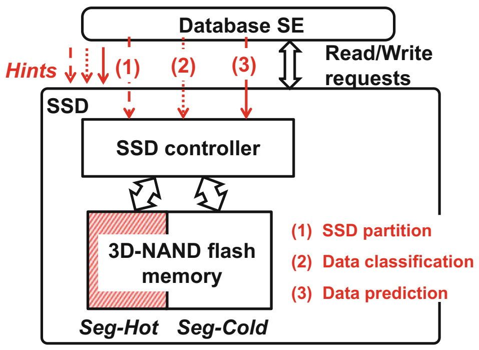

Fig. 12.7 SEA-SSD architecture [3]

Figure 12.7 presents the architecture of the SEA-SSD [3]. Each 3D-NAND flash chip is divided into two logical segments. *Seg-Hot* for the hot data (frequently accessed data) and *Seg-Cold* for the cold data (seldom accessed data). By aggregating data with similar activities in the same block, the GC overhead can be reduced. To determine the size of each segment, the first kind of hint is sent to the SSD controller, which is based on the strong correlation between the SE settings and hot data size. For the Innodb storage engine, the settings include the buffer pool and redo log sizes. The buffer pool caches the frequently accessed data (hot data) while redo log is used for crash recovery that guarantees the durability of the Innodb storage engine. The second hint is for data preliminary classification with a dynamic data model. If the data is judged as hot by the storage engine when it is flushed, logical "1" is sent to the SSD controller indicating that the data is hot and thus stored in the Seg-Hot, as shown in Fig. 12.8 [3]. Otherwise, it is simply stored in Seg-Cold. As the activities of the data stored in the 3D-NAND flash memory change with time, the data is predicted again with the third hint when the GC is triggered in the SSD. The third hint is the logical address of the page data that enters the flush list for the first time, since the data will be flushed to the SSD soon. As shown in Fig. 12.8 [3], such data should be stored in the Seg-Hot while other data are stored in Seg-Cold after the GC.

To evaluate the SEA-SSD, a database and SSD coupled simulator has been developed, which is over 20-times faster than the virtual platform, based on the Synopsys Platform Architect [9]. From the evaluation results, the write performance

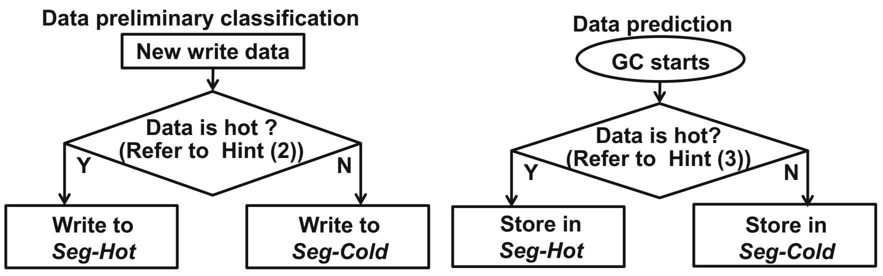

Fig. 12.8 SEA-SSD data management algorithms [3]

is improved by 24 % at maximum. Moreover, maximum 16 % energy consumption and 19 % lifetime enhancement are achieved.

12.3.2 Logical Block Address (LBA) Scrambled SSD

SEA-SSD is a design specially optimized for the database application. As a general solution to improve the write performance of the all 3D-NAND flash SSD for all applications, the logical block address (LBA) scrambled SSD is proposed [10]. A middleware LBA scrambler based on the address remapping technology is added to the existed SSD system.

The concept of the LBA scrambler is to reduce the page-copy overhead of the GC actively, as explained in Fig. 12.9 [10]. There are three kinds of pages in the SSD: valid pages, free pages and invalid pages. Among the valid pages, pages that still own free space are called *fragmented pages*. As mentioned in Sect. 12.2, all the valid pages in the next erase block have to be copied to the free space of another block, which leads to the degradation of the SSD write performance. Thus, LBA scrambler is proposed to write small data to the remaining free space of the fragmented pages in the next erase block actively. Due to the overwrite, all these fragmented pages in the next erase block become invalid and the data in the SSD become less fragmented.

Figure 12.10 illustrates the LBA scrambled SSD based computer system [10]. To achieve the address remapping, the LBA scrambler introduced another logical address called scrambled LBA (SLBA). After LBA scrambling, the data address SLBA is sent to the SSD controller. SSD controller writes data to a physical 3D-NAND flash page by the logical-to-physical table in the FTL (SSD controller).

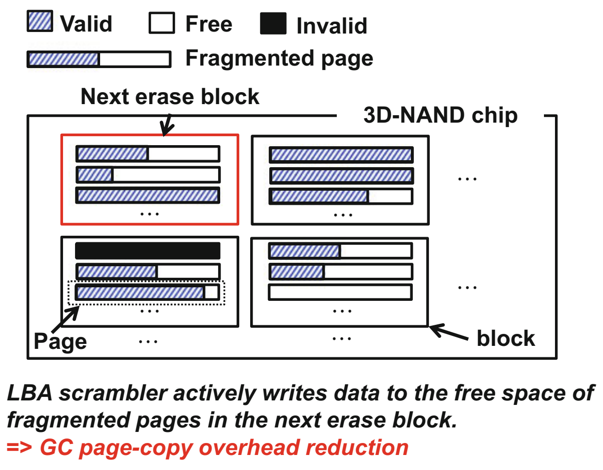
Valid
Free
Invalid
Fragmented page
Next erase block
3D-NAND chip
Page
block
LBA scrambler actively writes data to the free space of fragmented pages in the next erase block.
=> GC page-copy overhead reduction

Fig. 12.9 Concept of the LBA scrambler [10]

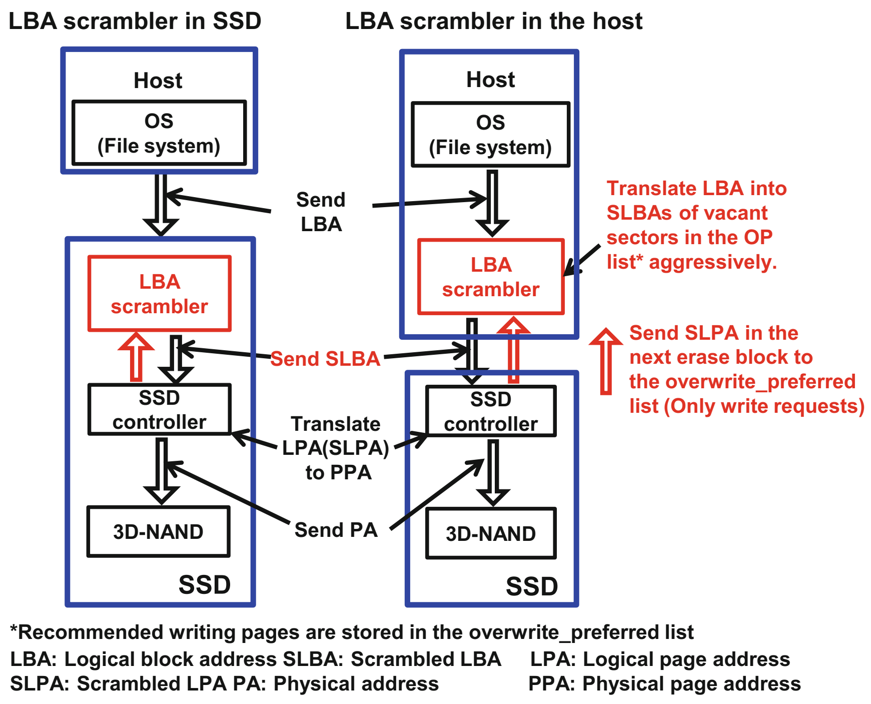
LBA scrambler in SSD
LBA scrambler in the host
Host
OS
(File system)
Send
LBA
LBA
scrambler
Send SLBA
SSD
controller
Translate
LPA(SLPA)
to PPA
3D-NAND
Send PA
SSD
Translate LBA into SLBAs of vacant sectors in the OP list* aggressively.
Send SLPA in the next erase block to the overwrite_preferred list (Only write requests)
*Recommended writing pages are stored in the overwrite_preferred list
LBA: Logical block address SLBA: Scrambled LBA
LPA: Logical page address
SLPA: Scrambled LPA PA: Physical address
PPA: Physical page address

Fig. 12.10 LBA scrambled SSD [10]

To inform the LBA scrambler about the fragmented page addresses, the scrambled logical page address (SLPA) of the fragmented page in the next erase block is sent to the LBA scrambler by the SSD controller. To record the address remapping between the LBA and SLBA, the LBA_to_SLBA table and unused_SLBA table are maintained in DRAM. As shown in Fig. 12.10, the LBA scrambler can locate in either the SSD or host [10]. When it locates in the SSD, a large DRAM capacity will be required for the SSD, but no interface modification is necessary. In contrast, a small DRAM capacity is required if LBA scrambler locates in the host. However, due to the communication between the LBA scrambler and the SSD controller, the interface of the SSD has to be upgraded. The algorithm flowchart of the LBA scrambler is shown in Fig. 12.11. For every N write requests, the hint (overwrite_preferred list) is updated with information transferred from the FTL to the LBA scrambler [10]. By referring to the overwrite_preferred list, the new write requests are generated for writing the fragmented pages in the next erase block actively. Moreover, the unaligned writes will create fragmented pages and additional overwrites as shown in Fig. 12.12. With the LBA scrambler's address remapping, the problem of unaligned writes for NAND flash memory can be eliminated.

From the evaluation results, maximum 394 % write performance improvement, 56 % energy consumption reduction and 55 % endurance enhancement are achieved with the LBA scrambler, compared with the SSD system without the LBA scrambler.

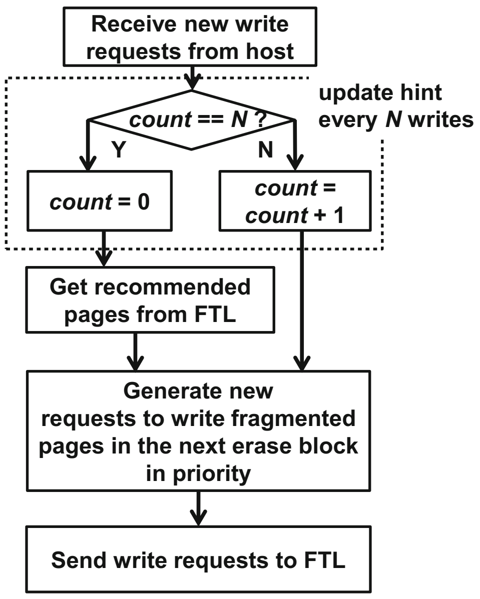
Receive new write requests from host
update hint every *N* writes
count == *N* ?
Y
N
count = 0
count = *count* + 1
Get recommended pages from FTL
Generate new requests to write fragmented pages in the next erase block in priority
Send write requests to FTL

Fig. 12.11 LBA scrambled algorithm flowchart [10]

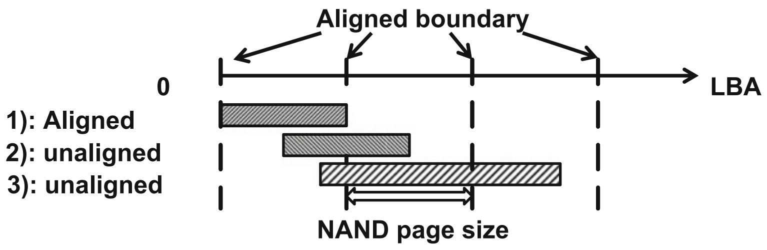

Fig. 12.12 Aligned and unaligned writes for the NAND flash memory

12.3.3 Hybrid M-SCM/3D-NAND Flash SSD

Both the SEA-SSD and LBA scrambled SSD adopts a middleware and SSD controller co-design methodology to improve the write performance of all 3D-NAND flash SSD by reducing SSD GC overhead. However, the write performance improvement of the SSD is limited by the read and write performance of the 3D-NAND flash memory.

On the other hand, SCM is much faster, more energy-efficient and endurable than 3D-NAND flash memory. It is non-volatile and supports in-place overwrite. Thus, both memory and storage systems are under a revolution due to SCM, as shown in Fig. 12.13 [11]. M-SCM is used in both memory and storage systems. In

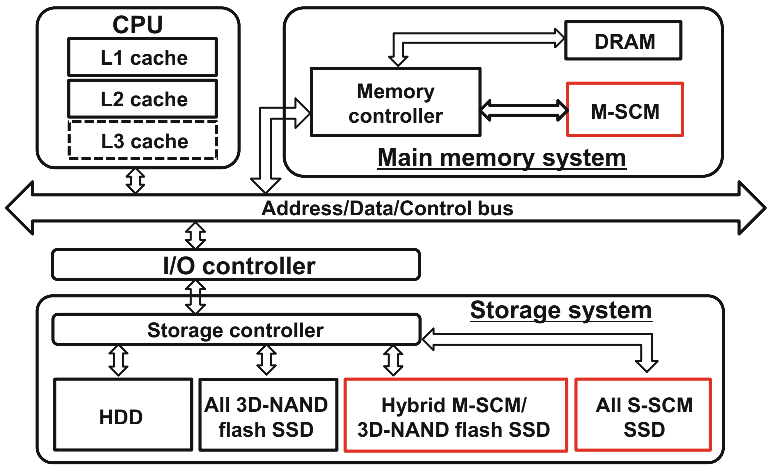

Fig. 12.13 Main memory and storage systems revolution due to SCM [11]

addition, S-SCM is used only in the storage system. By introducing SCM into the SSD system, the hybrid M-SCM/3D-NAND flash SSD is proposed to improve the write performance of the all 3D-NAND flash memory [12]. From the measurement results of the SCM (ReRAM) device, the verify cycles for write success vary with the write/erase (W/E) cycle. Thus, NAND-like interface with ready/busy status is adopted for the SCM. As shown in Fig. 12.3, M-SCM is used as a storage device for the SSD rather than a simple cache [13]. The data fragmentation suppression algorithm [12] and cold data eviction algorithm [13] are developed for the hybrid M-SCM/3D-NAND flash SSD considering both the data activity and data size. In the SSD controller, a least recently used (LRU) table is used to record the page

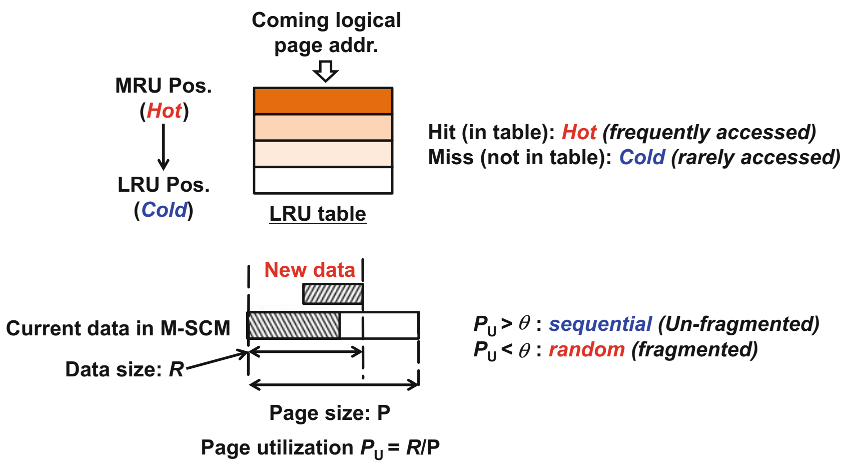
Coming logical page addr.
MRU Pos. (Hot)
LRU Pos. (Cold)
LRU table
Hit (in table): Hot (frequently accessed)
Miss (not in table): Cold (rarely accessed)
New data
Current data in M-SCM
Data size: R
Page size: P
Page utilization $P_U = R/P$
$P_U > \theta$: sequential (Un-fragmented)
$P_U < \theta$: random (fragmented)

Fig. 12.14 Criteria for data classification [11]

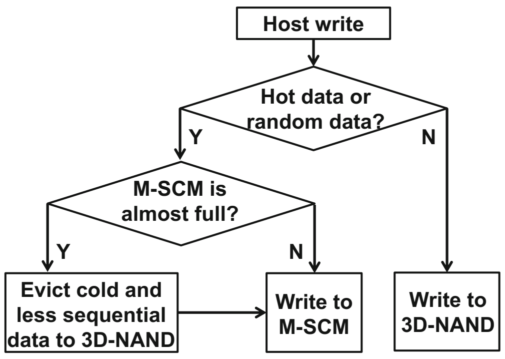

Fig. 12.15 Data management algorithm of the hybrid M-SCM/NAND flash SSD [13]

data's access history. As shown in Fig. 12.14, when the logical page address (LPA) of the page data hits the LRU, the page data is considered as hot (frequently accessed). Otherwise, the page data is judged as cold (seldom accessed). In addition, according to page utilization (data size divided by the page size), the page data are divided into two kinds: random (fragmented) and sequential (un-fragmented). When the page data size is over a threshold (θ), it is considered as the sequential data. The data storage policy of the hybrid SSD is to store hot data or random data in the M-SCM while cold and sequential data in the 3D-NAND flash. Hot data can update in-place and random data can accumulate and become sequential in M-SCM. The algorithm flowchart of the data management algorithm of the hybrid M-SCM/3D-NAND flash SSD is described in Fig. 12.15 [13]. When M-SCM becomes almost full, cold and less fragmented M-SCM data are evicted to the 3D-NAND flash. For the eviction procedure, the threshold to judge the data is sequential or random is a dynamic value, increasing/decreasing according to the data storage status in M-SCM. When it is hard to find the eviction candidates with the current threshold, the threshold is reduced to relax the restriction.

For the hybrid M-SCM/3D-NAND flash SSD, there are two important design considerations. Firstly, understand the M-SCM capacity and latency requirement for representative applications. Secondly, understand the effect of the 3D-NAND organization on the SSD write performance. Before the analyses, the SSD workloads are classified into four categories: hot and random, hot and sequential, cold and random, cold and sequential, as shown in Fig. 12.16 [14]. A large value of the average overwrite, defined as the total write data size divided by the user data size, indicates the workload is hot since it contains many hot data. In addition, the percentage of the random write request determines whether the workload is random or sequential. Here, half of the NAND flash page size is used as the threshold to judge the page is random or sequential.

From the evaluation results, increasing M-SCM capacity is more efficient to boost the write performance of the SSD than increasing the 3D-NAND flash overprovisioning (additional capacity over the user data size) with hot and random workload. Both increasing M-SCM capacity and 3D-NAND flash overprovisioning is capable of improving the SSD write performance. However, neither increasing

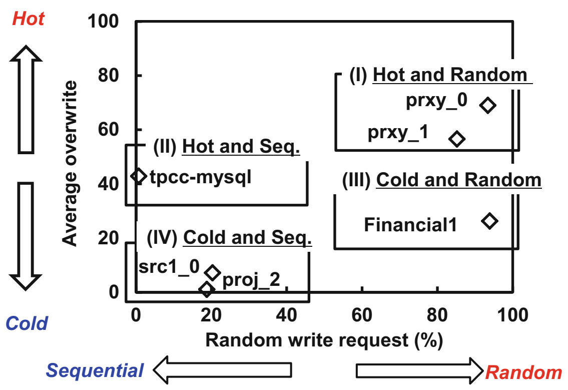

Fig. 12.16 SSD workload classification according to the data activity and size [14]

M-SCM capacity nor 3D-NAND flash overprovisioning is very effective for the write performance boost of the cold and sequential workload. Therefore, introducing M-SCM to the SSD is most suitable for the hot and random workload but not cost-efficient for the cold and sequential workload. Generally, less than 10 % M-SCM/3D-NAND flash capacity ratio is enough for the representative workloads with a fixed M-SCM latency of 100 ns/sector. On the other hand, a faster speed can be achieved by increasing the chip area for memory chip design. For example, write speed can be increased by enlarging the internal write unit and read speed can be improved by adding select devices for the reduction of bit line capacitance. When the speed of M-SCM is increased, the maximum throughput of the hybrid M-SCM/3D-NAND flash SSD is improved. As shown in Fig. 12.17, the write performance of the hybrid SSD saturates when the capacity of the M-SCM is over a threshold for a proxy server application (prxy_0), which is a hot and random intensive [11]. For other workloads such as Financial1, which is from a financial server, there is no trend of saturating when increasing the M-SCM/3D-NAND flash

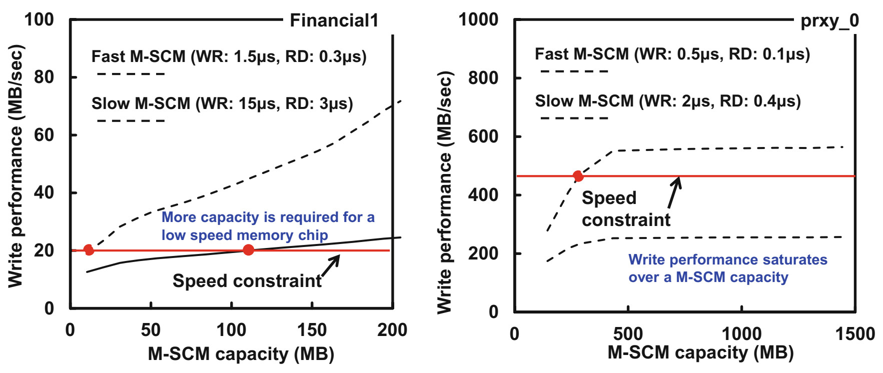

Fig. 12.17 SSD write performance dependency on the M-SCM capacity, latency and application [24]

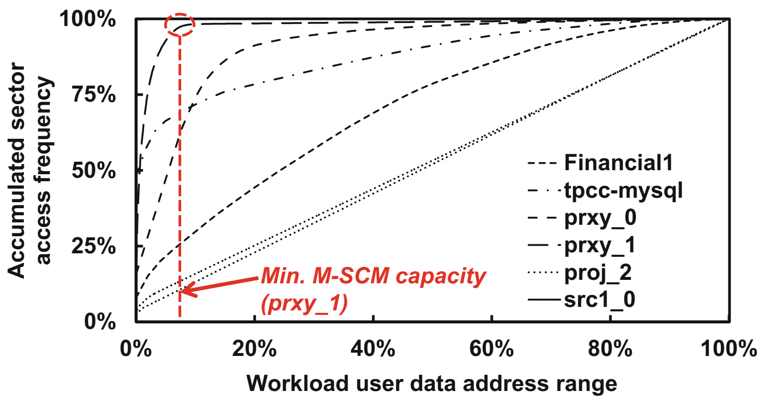

Fig. 12.18 Minimum M-SCM capacity requirement

capacity ratio. From the evaluation results of various workloads, the write performance of the hybrid M-SCM/3D-NAND flash SSD is workload/application dependent. Moreover, less M-SCM capacity is required for a faster speed M-SCM to reach the target application throughput. From the system point of view, there is a tradeoff between the M-SCM capacity requirement and M-SCM speed. Thus, there would be a cost-effective M-SCM chip design for a certain application, which is discussed in [11] by establishing optimistic and pessimistic SCM area cost models.

The minimum M-SCM capacity for the hybrid SSD is shown in Fig. 12.18 by analyzing the SSD workload. It illustrates the relationship between the accumulated sector write frequency and the address range of the write user data. For example, 25 % access frequency at 20 % user data address range means 25 % of the accesses occur at the top 20 % address of the user data. The turning point of the curve indicates the end of the frequently accessed data, usually random data, which requires high input output per second (IOPS). High slope value of the curve shows the most critical data, determining the minimum M-SCM capacity for the hybrid SSD. For Financial1 workload, M-SCM capacity should be over 40 % of the user data size to cover 75 % of the sector accesses. However, due to the temporal and spatial localities, the actual required SCM capacity as a write cache buffer is much smaller than 40 %. The rising trend of the curve is consistent with the results in Fig. 12.17. Increasing M-SCM capacity is effective to improve the hybrid SSD throughput for Financial1 workload. Further, as the slope value of the "Financial1" curve is smaller than that of prxy_0 and prxy_1, increasing SCM capacity is more effective for boosting the performance of the proxy server applications (prxy_0 and prxy_1). From Fig. 12.18, M-SCM capacity of less than 20 % of the user data size is enough for the proxy server application.

Since the conventional planar scaling of the NAND flash memory is facing various limitations making it harder and harder to reduce the fabrication cost and guarantee the memory reliability, 3D technology becomes a viable way to continue the trend of bit cost reduction for the NAND flash memory. Several 3D-NAND

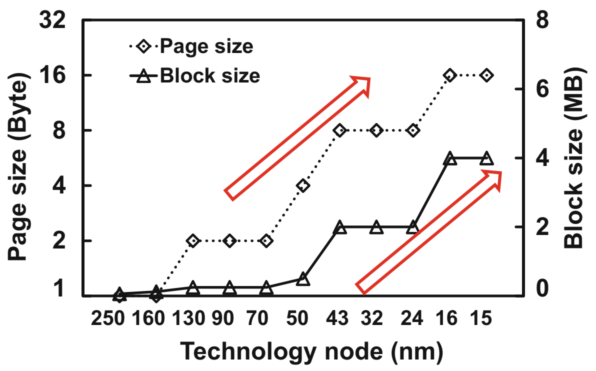

Fig. 12.19 Trend of NAND flash block and page sizes [1, 20–23]

flash architectures have been proposed. For example, terabit cell array transistor (TCAT) [15], Pipe-Shaped bit-cost scalable (P-BiCS) [16], vertical stacked array transistor (VSAT) [17], and dual control gate surrounding floating gate (DC-SF) [18]. There are two types of the 3D array: vertical channel and vertical gate. The current flows vertically and horizontally for the vertical channel type array and vertical gate array, respectively. 3D-NAND increases the bit density in the vertical direction (Z-dimension) in additional to the XY-dimensions. In case of the PiBCS 3D-NAND flash memory, the capacity of the 3D-NAND flash is increased by stacking more layers, which also compensates the problem of the reduced cell density in the XY-dimensions due to the non-scalable BiCS hole's diameter [19].

For design of the 3D-NAND flash memory, the NAND organization is critically important to the performance and cost of the circuits. A group of NAND flash memory cells is connected in series as a NAND flash string. Multiple NAND flash strings sharing the same substrate consists a NAND flash block as the erase unit. By asserting a high voltage on the substrate to eject the electrons from the floating gate of the memory cells in the block, the erase operation is executed. In the block, the memory cells connected with the same word-line is a page. It is the read/write unit. With the scaling, there is an increasing trend for the page and block sizes of the

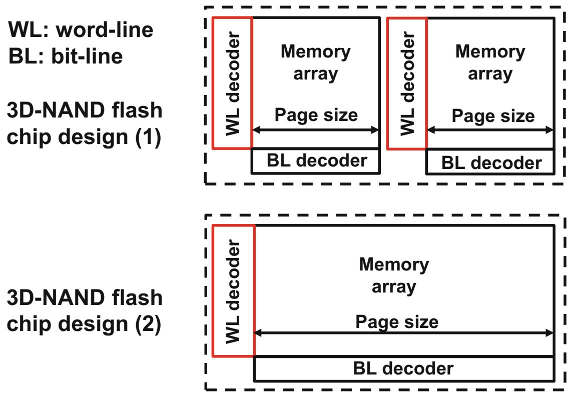

Fig. 12.20 Chip design of the 3D-NAND flash memory [24]

NAND flash memory, as shown in Fig. 12.19 [1, 20–23]. The typical size of the NAND flash page size is 8 kB and block size is 2 MB (256 pages in a block). With the advent of 3D-NAND flash memory, larger page and block sizes can be easily adopted. Take P-BiCS 3D-NAND for example, the block size of the NAND flash is doubled by doubling the stack layers. On the other hand, as shown in Fig. 12.20, adopting a large page size design reduces the word-line decoder area overhead of the NAND flash memory chip compared to the design of adopting a small page size [24]. However, it is not true that larger page or block size is better for the performance of the real-world applications.

Figure 12.21 shows the evaluation results of the block size sensitivity analysis for the all 3D-NAND flash memory [24]. The page size is fixed to 16 kB. Take

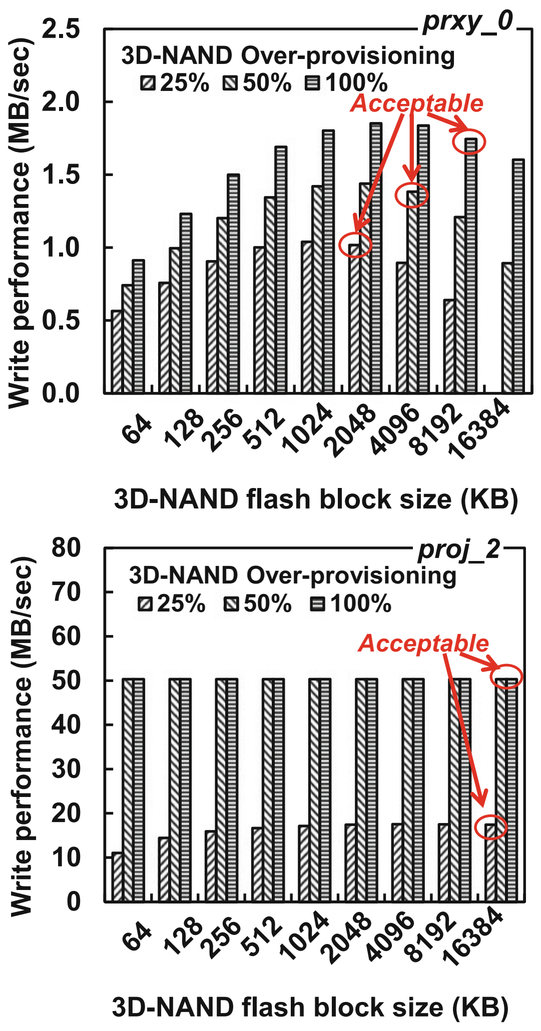

Fig. 12.21 Block size evaluation of the all 3D-NAND flash memory [24]. 2 MB is the typical block size [25]

prxy_0, a firewall/web proxy server workload for example, there is a maximum value for the write performance of the all 3D-NAND flash SSD with a certain capacity. Too small or too large block size decreases the write performance. Large block size may induce long GC latency while small block size will reduce the erase throughput. Assuming 10 % write performance is tolerable for the all 3D-NAND flash SSD, the acceptable block sizes for the all 3D-NAND flash memory with 25, 50 and 100 % over-provisioning are 2, 4 and 8 MB, respectively. Higher all 3D-NAND flash SSD capacity accepts a larger block size. Moreover, for the proj_2, a project directories sever workload (cold and sequential), the write performance saturates when the block size is over a threshold. Even the block size is as large as 16 MB, there is no write performance degradation, which is great for the application of 3D-NAND flash. It is because such workload seldom leads to the triggering of the GC. The cold and sequential data just fill in the block.

The analyses of page size sensitivity of the all 3D-NAND flash SSD are presented in Fig. 12.22 [24]. The block size is fixed to 4 MB. Similar to the block size sensitivity, over large page size or small page size degrades the write performance

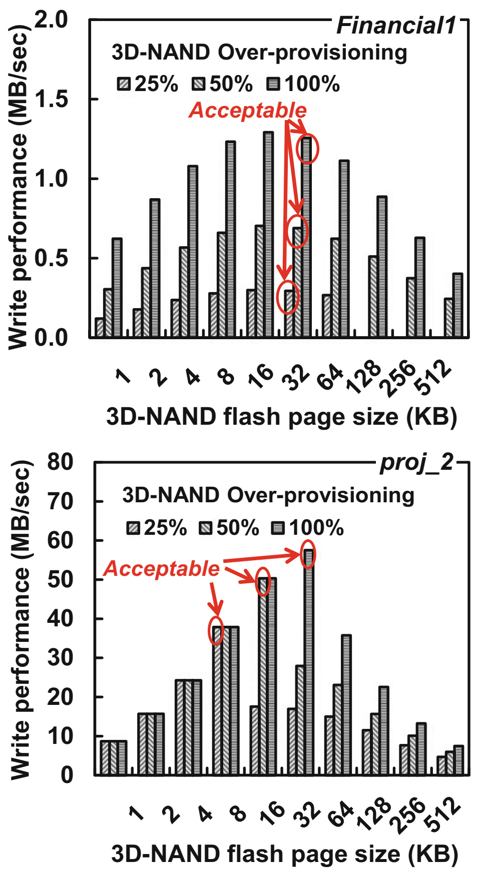

Fig. 12.22 Page size evaluation of the all 3D-NAND flash memory [24]. 8 kB is the typical page size [25]

of the all 3D-NAND flash SSD. Large page size is good for the sequential write throughput but the page overwrite count would be large (more page overwrite overhead). In addition, fewer pages exist in the case of larger page size. Thus, the GC will be triggered more frequently. In contrast, small page size induces less page overwrite overhead but it is not good for sequential writes. Further, more pages may need to be copied during GC. From the experimental results, for workloads like Financial1 a financial online transaction processing (OLTP) sever workload, the acceptable page sizes are the same at 25, 50 and 100 % SSD capacity overprovisioning cases. Only for proj_2 that is cold and sequential, larger page size is acceptable for a larger SSD capacity over-provisioning.

Moreover, larger block and page sizes are acceptable for the M-SCM/3D-NAND flash hybrid SSD, as shown in Figs. 12.23 and 12.24 [24]. Fix the page size as 16 kB

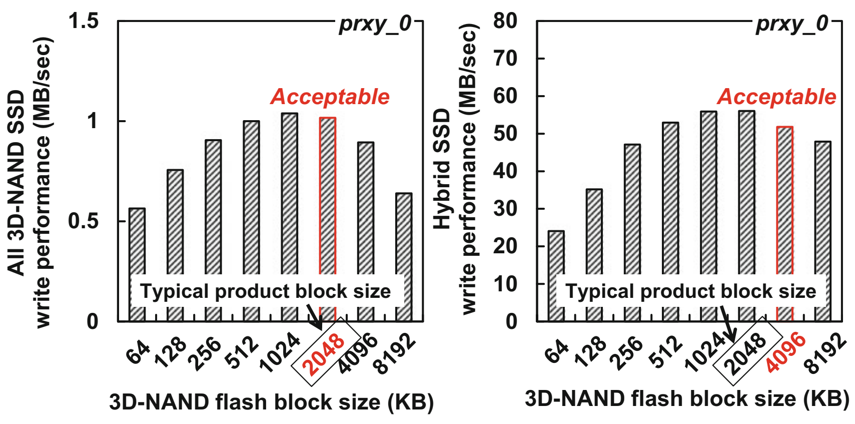

Fig. 12.23 Block size evaluation of the hybrid M-SCM/3D-NAND flash memory [24]

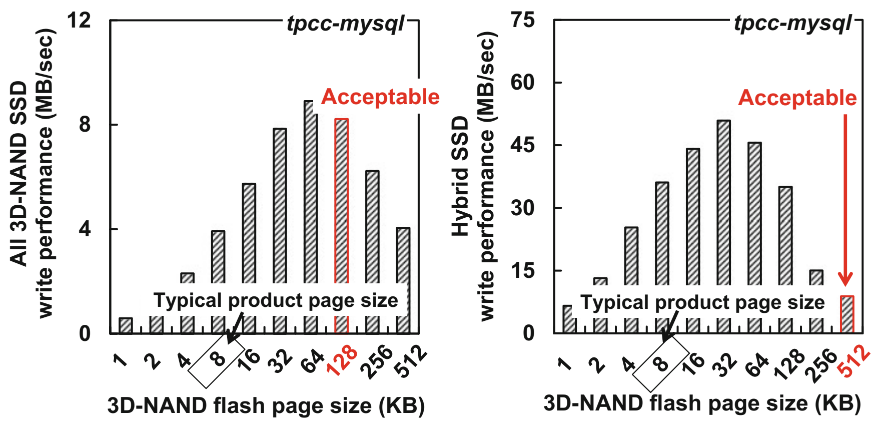

Fig. 12.24 Page size evaluation of the hybrid M-SCM/3D-NAND flash memory [24]

and 3D-NAND flash over-provisioning as 25 % for the prxy_0 workload. The acceptable block size is 2 MB in the case of the all 3D-NAND flash SSD, which is the same size of the current typical NAND flash block size [25]. In the case of the hybrid M-SCM/3D-NAND flash SSD, the M-SCM/3D-NAND flash capacity ratio is set as 8.5 %. Assuming the bottom line of the design of the hybrid SSD is its write performance should be larger than that of the all 3D-NAND flash SSD, the acceptable block size of the hybrid SSD is 4 MB, which is 2 times that of the 3D-NAND flash acceptable block size. It indicates that with M-SCM, the stacking layers of the 3D-NAND flash could be doubled for the prxy_0 workload. As shown in Fig. 12.24, the typical page size of the NAND flash product is 8 kB [25]. The acceptable page sizes for the all 3D-NAND flash SSD and hybrid M-SCM/3D-NAND flash SSD are 128 and 512 kB, respectively, with the tpcc-mysql (a relational database workload).

The comparison results of the 3D-NAND flash design for the all 3D-NAND flash SSD and hybrid M-SCM/3D-NAND flash SSD are summarized in Fig. 12.25 [24]. With M-SCM, the acceptable block and page sizes are enlarged by 4 times and 64 times, respectively. The 3D-NAND flash stacking layers could be quadruple for the hybrid SSD compared with the all 3D-NAND flash SSD, without any write performance degradation.

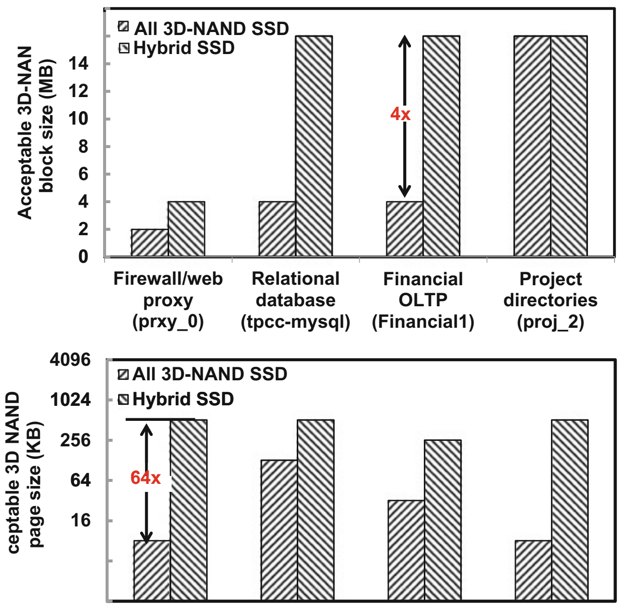

Fig. 12.25 Comparison of the acceptable 3D-NAND flash page and block sizes for all 3D-NAND flash SSD and hybrid M-SCM/3D-NAND SSD [24]

12.3.4 All S-SCM SSD

When the SCM technology matures and the cost is reduced to be competitive to that of the NAND flash memory, the all S-SCM SSD becomes a viable solution to replace the current NAND flash-based SSD. In this section, we present the wear leveling, S-SCM I/O data toggle rate, S-SCM latency design for the all S-SCM SSD.

For S-SCM candidates like ReRAM, the device endurance is limited (10^7 for 50 nm HfO2 ReRAM [26]), although it is high compared with the NAND flash memory. Therefore, wear leveling is required for S-SCM. A simple wear leveling algorithm is shown in Fig. 12.26, which is operated in a page-level. Thus, a wear leveling triggering threshold δ is maintained for each page of the S-SCM. Further, to monitor the endurance of each sector (512 Byte), that is the minimum access unit in the block device, the write/erase (W/E) cycle for each sector is maintained. When the maximum W/E cycle of the sectors in the page *i* is smaller than $\delta_{\mathrm{page}(i)}$, in-place page overwrite is executed. Otherwise, the wear leveling is triggered. During the wear leveling, the data in the old page *i* is read out, merged with the new data and written to a new page *j*. After that, the wear leveling triggering threshold of page *i* is updated with a constant window threshold σ, to raise the bar of the wear leveling triggering. Actually, there are many hot spots (some addresses are frequently

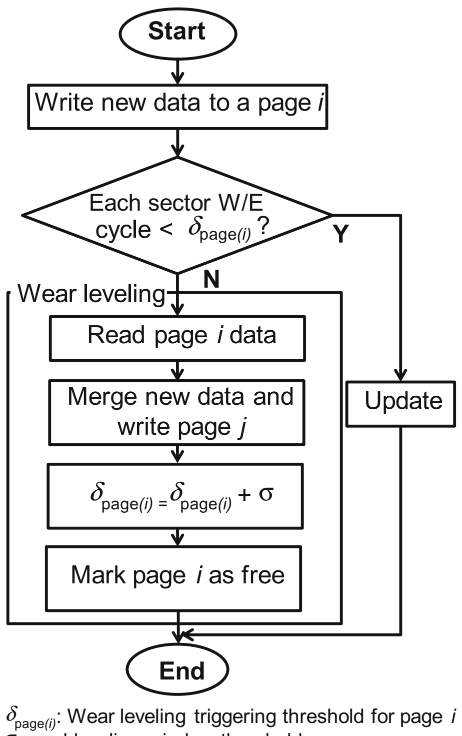
Start
Write new data to a page i
Each sector W/E cycle < $\delta_{\mathrm{page}(i)}$?
Y
N
Wear leveling
Read page i data
Merge new data and write page j
$\delta_{\mathrm{page}(i)} = \delta_{\mathrm{page}(i)} + \sigma$
Mark page i as free
Update
End
$\delta_{\mathrm{page}(i)}$: Wear leveling triggering threshold for page *i*
σ: weal leveling window threshold

Fig. 12.26 Wear leveling algorithm flowchart for all S-SCM SSD [27]

written) in the workload, the endurance of the all S-SCM SSD can be greatly enhanced with the wear leveling. For instance, with σ = 5, the maximum W/E cycle of the sector in the S-SCM without the wear leveling is over 3000 times higher than that of the all S-SCM SSD with wear leveling procedure.

By adjusting the value of σ, the wear leveling triggering interval can be controlled. A small σ triggers the wear leveling easily to make all pages wear out evenly. However, such configuration would degrade the all S-SCM SSD performance due to the additional page-copy operations. Therefore, σ could be adjusted to balance the SSD performance and endurance [27].

Figure 12.27 shows the all S-SCM SSD write performance dependency on the I/O data toggle rate, S-SCM latency (assuming the some read and write latency) and applications [27]. M-SCM latency is set as 100 ns/sector. Different from the 3D-NAND flash based SSD, there is little write performance dependency on the applications. The trend is the same. The slight difference is due to the S-SCM wear leveling and total write data size. When S-SCM speed is faster, a higher data toggle rate should be adopted to fully exploit the write performance of the all S-SCM SSD. By keeping S-SCM latency as 1 μs, the speeds of the all S-SCM SSD and hybrid M-SCM/3D-NAND flash SSD are compared in Fig. 12.28a, which illustrates the breakpoint of the I/O data toggle rate at 25 % fixed M-SCM/3D-NAND flash ratio [27]. Take tpcc-mysql workload for example, over 500 MHz I/O data toggle rate makes the all S-SCM SSD faster than the hybrid M-SCM/3D-NAND flash SSD. On the other hand, at a fixed I/O data toggle rate of 1066 MHz, the breakpoint of the S-SCM latency can be analyzed, as shown in Fig. 12.28b [27]. Faster S-SCM

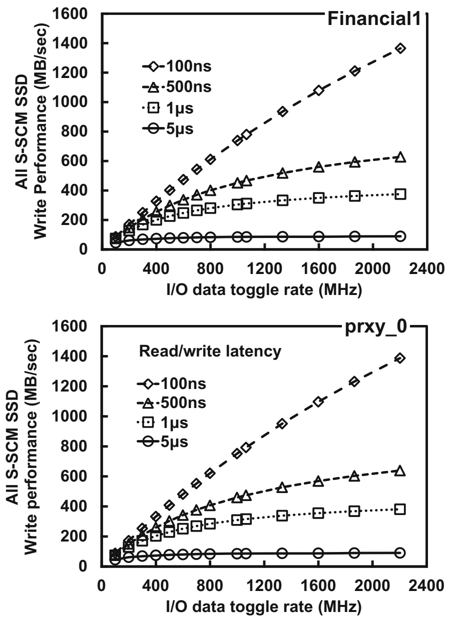

Fig. 12.27 All S-SCM SSD write performance dependency on the I/O data toggle rate, S-SCM latency and applications [27]

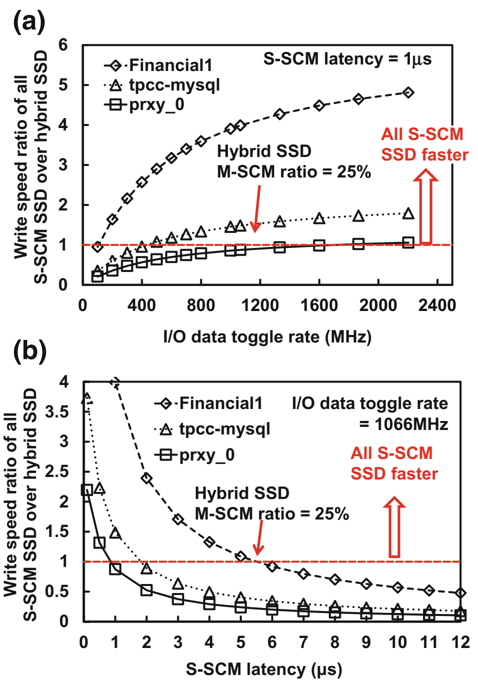

Fig. 12.28 Write speed comparison of the all S-SCM SSD and hybrid M-SCM/3D-NAND flash SSD [27]

device creates a faster all S-SCM SSD. In the case of the Financial1 workload, the hybrid M-SCM/3D-NAND flash SSD owns a faster speed than the all S-SCM SSD if the S-SCM latency is over 5 µs.

Moreover, the cost of each SSD can be compared easily according to the capacity configurations of the memories inside the SSD, by assuming the bit cost of each memory device.

It is interesting to know how speedy storage system can be achieved by SCM, compared with the NAND flash. Expression (12.1) shows the calculation of the latency of a page write operation T_{NAND}, where $T_{\mathrm{CMD \cdot N}}$ is the latency of issuing the program command and programming addresses, $T_{\mathrm{IO \cdot N}}$ is the time of loading the data into the data register of the NAND flash memory, $T_{\mathrm{MEM \cdot N}}$ is the time of storing the data from the data register to the memory array (array programming time).

$$T_{\mathrm{NAND}} = T_{\mathrm{CMD \cdot N}} + T_{\mathrm{IO \cdot N}} + T_{\mathrm{MEM \cdot N}} \tag{12.1}$$

$T_{\mathrm{CMD \cdot N}}$ is only a few clocks long. Compared with $T_{\mathrm{IO \cdot N}}$ and $T_{\mathrm{MEM \cdot N}}$, it is negligible small. $T_{\mathrm{IO \cdot N}}$ is inversely proportional to the data toggle rate $P_{\mathrm{Toggle \cdot N}}$, defined by the NAND flash interface, as shown in formula (12.2), where L_{NAND} is the NAND flash page size and $W_{\mathrm{DAT \cdot N}}$ is the width of the data bus.

$$T_{\mathrm{IO \cdot N}} = \frac{L_{\mathrm{NAND}}}{W_{\mathrm{DAT \cdot N}}} \times \frac{1}{P_{\mathrm{Toggle \cdot N}}} \tag{12.2}$$

When the data toggle rate $P_{\mathrm{Toggle \cdot N}}$ is high, $T_{\mathrm{IO \cdot N}}$ is reduced. Usually, $T_{\mathrm{IO \cdot N}}$ should be much smaller than $T_{\mathrm{MEM \cdot N}}$. If $T_{\mathrm{IO \cdot N}}$ is comparable or even higher than $T_{\mathrm{MEM \cdot N}}$, the interface becomes the memory device performance bottleneck. On the other hand, a page cache program may be supported by the NAND flash memory, which uses the internal cache register to improve the NAND flash program throughput, as shown in Fig. 12.29. During the 1st program data storing from the data register to the memory array, the 2nd program data can be input to the page cache. By the page cache, the $T_{\mathrm{CMD \cdot N}}$ and $T_{\mathrm{IO \cdot N}}$ are hidden. As a result, the latency of writing N NAND flash pages T_{N} can be calculated by (12.3) and (12.4).

$$T_{\mathrm{N}} = T_{\mathrm{CMD \cdot N}} + T_{\mathrm{IO \cdot N}} + N \times T_{\mathrm{MEM \cdot N}} \tag{12.3}$$

$$T_{\mathrm{N}} = T_{\mathrm{CMD \cdot N}} + T_{\mathrm{IO \cdot N}} + \left\lceil \frac{L_{\mathrm{DAT}}}{L_{\mathrm{NAND}}} \right\rceil \times T_{\mathrm{MEM \cdot N}} \tag{12.4}$$

Note that (12.4) is true only when the $T_{\mathrm{MEM \cdot N}}$ is much longer than the $T_{\mathrm{CMD \cdot N}}$ and $T_{\mathrm{IO \cdot N}}$. When the $T_{\mathrm{IO \cdot N}}$ is large, the page cache cannot completed be hidden. Moreover, the page cache does not work in the random write case. Without page cache effect, the sequential write latency of the NAND flash memory is expressed by (12.5):

$$T_{\mathrm{N}} = \left\lceil \frac{L_{\mathrm{DAT}}}{L_{\mathrm{NAND}}} \right\rceil \times \left(T_{\mathrm{CMD \cdot N}} + T_{\mathrm{IO \cdot N}} + T_{\mathrm{MEM \cdot N}}\right) \tag{12.5}$$

Additionally, the sequential write throughput ratio of the SCM device is expressed by (12.6), where the write unit of SCM is L_{SCM}. $T_{\mathrm{CMD \cdot S}}$ is the latency of

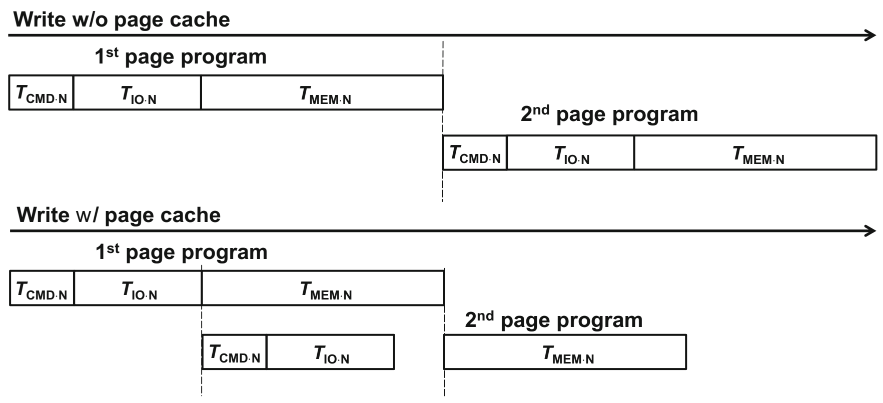
Write w/o page cache
1st page program
$T_{\mathrm{CMD \cdot N}}$
$T_{\mathrm{IO \cdot N}}$
$T_{\mathrm{MEM \cdot N}}$
2nd page program
$T_{\mathrm{CMD \cdot N}}$
$T_{\mathrm{IO \cdot N}}$
$T_{\mathrm{MEM \cdot N}}$
Write w/ page cache
1st page program
$T_{\mathrm{CMD \cdot N}}$
$T_{\mathrm{IO \cdot N}}$
$T_{\mathrm{MEM \cdot N}}$
2nd page program
$T_{\mathrm{CMD \cdot N}}$
$T_{\mathrm{IO \cdot N}}$
$T_{\mathrm{MEM \cdot N}}$

Fig. 12.29 Page cache operation of the NAND flash memory

issuing the program command and programming addresses, $T_{\mathrm{IO \cdot S}}$ is the time of loading the data into the data register of the SCM, $T_{\mathrm{MEM \cdot S}}$ is the time of storing the data from the data register to the memory array (array programming time). Since SCM is fast, page cache is not designed.

$$T_{\mathrm{S}} = \left\lceil \frac{L_{\mathrm{DAT}}}{L_{\mathrm{SCM}}} \right\rceil \times (T_{\mathrm{CMD \cdot S}} + T_{\mathrm{IO \cdot S}} + T_{\mathrm{MEM \cdot S}}) \tag{12.6}$$

According to formulas (12.2), (12.4) and (12.6), the SCM and NAND flash device write performance ratio $R_{\mathrm{S/N}}$ can be *estimated* by expression (12.7) without the considerations of the GC and overwrites in NAND flash only SSD and wear leveling in All SCM SSD. $W_{\mathrm{DAT \cdot S}}$ is the memory bus width of SCM. $P_{\mathrm{Toggle \cdot S}}$ is the data toggle rate of the SCM.

$$\begin{aligned} R_{\mathrm{S/N}} &= \frac{T_{\mathrm{CMD \cdot N}} + T_{\mathrm{IO \cdot N}} + \left\lceil \frac{L_{\mathrm{DAT}}}{L_{\mathrm{NAND}}} \right\rceil \times T_{\mathrm{MEM \cdot N}}}{\left\lceil \frac{L_{\mathrm{DAT}}}{L_{\mathrm{SCM}}} \right\rceil \times (T_{\mathrm{CMD \cdot S}} + T_{\mathrm{IO \cdot S}} + T_{\mathrm{MEM \cdot S}})} \\ &\approx \frac{\frac{L_{\mathrm{NAND}}}{W_{\mathrm{DAT \cdot N}}} \times \frac{1}{P_{\mathrm{Toggle \cdot N}}} + \left\lceil \frac{L_{\mathrm{DAT}}}{L_{\mathrm{NAND}}} \right\rceil \times T_{\mathrm{MEM \cdot N}}}{\left\lceil \frac{L_{\mathrm{DAT}}}{L_{\mathrm{SCM}}} \right\rceil \times \left(\frac{L_{\mathrm{SCM}}}{W_{\mathrm{DAT \cdot S}}} \times \frac{1}{P_{\mathrm{Toggle \cdot S}}} + T_{\mathrm{MEM \cdot S}}\right)} \end{aligned} \tag{12.7}$$

Assuming $T_{\mathrm{MEM \cdot N}}$ = 1.6 ms, L_{NAND} = 16,384 Bytes (32 sectors), L_{SCM} = 512 Bytes (1 sector), $P_{\mathrm{Toggle \cdot N}}$ = 400 Mbps/pin, $P_{\mathrm{Toggle \cdot S}}$ = 1066 Mbps/pin, $W_{\mathrm{DAT \cdot N}}$ = 1 Byte and $W_{\mathrm{DAT \cdot S}}$ = 1 Byte. The relationship of $R_{\mathrm{S/N}}$ and $T_{\mathrm{MEM \cdot S}}$ is shown in Fig. 12.30. From Fig. 12.30, it can be found that single SCM chip can achieve over 1000-times performance gain than the single NAND flash chip if SCM write latency is below 1 μs. For random writes ($L_{\mathrm{DAT}} < L_{\mathrm{NAND}}$), $R_{\mathrm{S/N}}$ is equivalent to the IOPS ratio of the SCM over the NAND flash memory. IOPS is used as the metric of

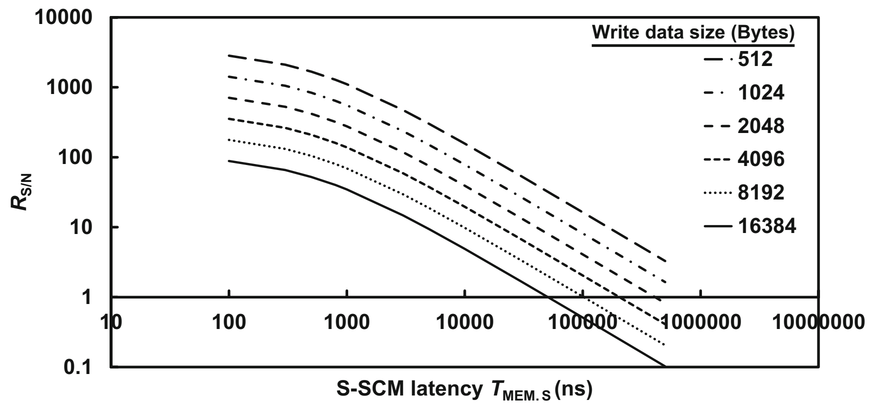

Fig. 12.30 S-SCM and 3D-NAND flash memory throughput ratio

the random write performance. If SCM has a latency of 100 µs, $R_{S/N}$ is less than 100 at any L_{DAT}. Furthermore, $R_{S/N}$ becomes less than 1 when the $L_{DAT} > L_{NAND}$. Thus, such a long latency SCM device is not cost-efficient. The curves almost overlap in Fig. 12.30 at the condition of $L_{DAT} > L_{NAND}$ (sequential write), which shows the lowest $R_{S/N}$ that the SCM can achieve. Below 50 µs, the SCM still has a higher performance than the NAND flash memory. However, the SCM is slower than the NAND flash memory if its latency is over 50 µs. It indicates that the NAND flash is good for the sequential write, thanks to the low write energy of the memory cell. In other words, achieving over 1000-times sequential write performance boost (>1 NAND flash page size) for SCM device is extremely difficult.

A recent ReRAM device [28] and MLC NAND flash memory [29] presented in ISSCC 2014 have the program latencies 10 µs/2048 Bytes and 1185 µs/16 kB, respectively. Comparing the SSDs made of these two devices, the performance gain of the all ReRAM SSD with 512 B random data pattern can be over 100-times (over 50-times at 4 kB random data pattern). To improve the system performance gain, the SCM chip latency should be further reduced or the device program current has to be reduced for increasing the number of parallel program cells (corresponding to L_{SCM}).

For the enterprise applications, the random data access is the bottleneck. Since the real workload is the mixture of random and sequential data, MB/s is still used as the overall performance metric throughout the chapter. The average IOPS $IOPS_{Avg}$ can be calculated by formula (12.8):

$$IOPS_{AVG} = \frac{Throughput_{AVG} \times 1024}{RS_{AVG}} \tag{12.8}$$

where RS_{AVG} is the average request size (kB) and $Throughput_{AVG}$ is the average SSD performance (MB/s). If the SSD average write performance is 20 MB/s with the 512 Byte random write data pattern, the average IOPS is equal to 40,960. When the SSD is tested with 4 kB random data, the average SSD IOPS is 5120.

12.4 Summary and Conclusion

Three techniques are presented in this chapter to improve the write performance of the 3D-NAND flash-based SSD. Table 12.1 summarizes the pros and cons of the techniques in this chapter. The SEA-SSD and LBA scrambled SSD are short-term solutions, which co-design the middleware and SSD controller. To overcome the hurdle of limited write performance of the NAND flash memory, hybrid M-SCM/3D-NAND flash memory SSD is the mid solution which is able to improve the conventional all 3D-NAND flash SSD write performance by over 10 times. In the long run, all S-SCM SSD is promising. With the system proposals like the SEA-SSD and LBA scrambler, the latency design parameters for the 3D-NAND flash memory are relieved. On the other hand, a larger page size and block size can

Table 12.1 3D-NAND flash design with system-level considerations

Solution	Pros	Cons	Implications for 3D-NAND flash design
SEA-SSD [3] (short-term)	• Low cost • Use upper layer information • Low data management complexity	• Only for database application	• Relaxed latency and endurance design constraints for 3D-NAND flash based SSD by integrating these techniques • Larger page and block sizes are acceptable for hybrid M-SCM/3D-NAND flash SSD, compared with the all 3D-NAND flash SSD • Customized 3D-NAND flash design enables write performance optimization for each application
LBA scrambler [10] (short-term)	• Low cost • Eliminate page fragmentation due to unaligned writes • Enhance write speed for all representative SSD workloads	• More DRAM capacity is required for tables • Interface may need to be modified	
Hybrid M-SCM/3D-NAND flash hybrid SSD [13] (mid-term)	• Cost-effective (compared to all M-SCM SSD) • Improve SSD write speed, power and reliability greatly	• Complicated tiered storage/cache algorithm is required	
All S-SCM SSD [27] (long-term)	• Little write performance dependency on application • In-place overwrite • No garbage collection is required • >100 times SSD write speed boost is possible	• High cost (currently) • S-SCMs like ReRAM are still in early development stage	

be accepted for the NAND flash memory by introducing M-SCM into the SSD, which is good for the design of the 3D-NAND flash memory. Due to the write performance dependency on the application, custom 3D-NAND flash design enables the performance optimization for each application.

References

1. K. Takeuchi, Novel co-design of NAND flash memory and NAND flash controller circuits for sub-30 nm low-power high-speed solid-state drives (SSD). IEEE J. Solid-State Circ. **44**(4), 1227–1234 (2009)
2. K. Takeuchi et al., A 56 nm CMOS 99 mm^2 8 Gb multi-level NAND flash memory with 10-MB/s program throughput. IEEE J. Solid-State Circ. **42**(1), 219–232 (2007)
3. C. Sun et al., SEA-SSD: a storage engine assisted SSD with application-coupled simulation platform. IEEE Trans. Circ. Syst. I **62**(1), 120–129 (2015)
4. FusionIO, http://www.fusionio.com/press-releases/fusion-io-software-development-kit-enables-native-flash-memory-access
5. A.M. Caulfield et al., Moneta: a high-performance storage array architecture for next-generation, non-volatile memories, in *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2010), pp. 385–395
6. A.M. Caulfield et al., Providing safe, user space access to fast, solid state disks, in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2012), pp. 387–400
7. A. Trivedi et al., Unified high-performance I/O: one stack to rule them all, in *Proceedings of the USENIX Workshop on Hot Topics in Operating Systems (HotOS)* (2013)
8. S. Peter et al., Towards high-performance application-level storage management, in *Proceedings of the Workshop on Hot Topics in Storage and File System (HotStorage)* (2014)
9. Synopsys Platform Architect, http://www.synopsys.com/Systems/ArchitectureDesign/Pages/PlatformArchitect.aspx
10. C. Sun et al., LBA scrambler: a NAND flash aware data management scheme for high-performance solid-state drives. IEEE Trans. Very Large Scale Integr. (VLSI) Syst. (2015 in press)
11. C. Sun et al., Cost, capacity and performance analyses for hybrid SCM/NAND flash SSD. IEEE Trans. Circ. Syst. I **61**(8), 2360–2369 (2014)
12. H. Fujii et al., x11 performance increase, x6.9 endurance enhancement, 93 % energy reduction of 3D TSV-integrated hybrid ReRAM/MLC NAND SSDs by data fragmentation suppression, in *IEEE Symposium on VLSI Circuits* (2012), pp. 134–135
13. C. Sun et al., A high performance and energy-efficient cold data eviction algorithm for 3D-TSV hybrid ReRAM/MLC NAND SSD. IEEE Trans. Circ. Syst. I **61**(2), 382–392 (2014)
14. C. Sun et al., SCM capacity and NAND over-provisioning requirements for SCM/NAND flash hybrid enterprise SSD, in *Proceedings on International Memory Workshop (IMW)* (2013), pp. 64–67
15. J. Jang et al., Vertical cell array using TCAT (Terabit Cell Array Transistor) technology for ultra high density NAND flash memory, in *IEEE Symposium on VLSI Technology* (2009), pp. 192–193
16. R. Katsumata et al., Pipe-shaped BiCS flash memory with 16 stacked layers and multi-level-cell operation for ultra high density storage devices, in *IEEE Symposium on VLSI Technology* (2009), pp. 136–137
17. J. Kim et al., Novel Vertical-Stacked-Array-Transistor (VSAT) for ultra-high-density and cost-effective NAND flash memory devices and SSD (Solid State Drive), in *IEEE Symposium on VLSI Technology* (2009), pp. 186–187
18. S.J. Whang et al., Novel 3-dimentional dual control-gate with surrounding floating-gate (DC-SF) NAND flash cell for 1 Tb file storage application, in *IEEE International Electron Devices Meeting (IEDM)* (2010), pp. 668–671
19. K. Miyaji et al., Control gate length, spacing, channel hole diameter, and stacked layer number design for bit-cost scalable-type three-dimensional stackable NAND flash memory. Jpn. J. Appl. Phys. **53**, 024201 (2014)

20. C. Lee et al., A 32 Gb MLC NAND-flash memory with Vth-endurance-enhancing schemes in 32 nm CMOS, in *Proceedings IEEE International Solid-State Circuits Conference (ISSCC)*, Feb 2010, pp. 446–447
21. K. Fukuda et al., A 151 mm^2 64 Gb MLC flash memory in 24 nm cmos technology, in *Proceedings on IEEE International Solid-State Circuits Conference (ISSCC)*, Feb 2011, pp. 198–199
22. S. Choi et al., A 93.4 mm^2 64 Gb MLC NAND-flash memory with 16 nm cmos technology, in *Proceedings on IEEE International Solid-State Circuits Conference (ISSCC)*, Feb 2014, pp. 328–329
23. M. Sako et al., A low-power 64 Gb MLC NAND-flash memory in 15 nm cmos technology, in *Proceedings on IEEE International Solid-State Circuits Conference (ISSCC)*, Feb 2014, pp. 128–129
24. C. Sun et al., A workload-aware-design of 3D-NAND flash memory for enterprise SSDs, in *International Symposium on Quality Electronic Design (ISQED)*, March 2014, pp. 554–561
25. Micron Technology Inc., MT29512G08CUCAB data sheet, Nov 2009, http://micron.com
26. K. Higuchi et al., Evaluation of voltage vs. pulse width modulation and feedback during set/reset verify-programming to achieve 10 million cycles for 50 nm HfO2 ReRAM. Solid-State Electron. **91**, 67–73 (2014)
27. T. Onagi et al., Design guidelines of storage class memory based solid-state drives to balance performance, power, endurance and cost. Jpn. J. Appl. Phys. (JJAP) (2015 in press)
28. R. Fackenthal et al., A 16 Gb ReRAM with 200 MB/s write and 1 GB/s read in 27 nm technology, in *Proceedings on IEEE International Solid-State Circuits Conference (ISSCC)*, Feb 2014, pp. 338–339
29. M. Helm et al., A 128 Gb MLC NAND-flash device using 16 nm planar cell, in *Proceedings on IEEE International Solid-State Circuits Conference (ISSCC)*, Feb 2014, pp. 326–327

Index