Optimization of Multi-Channel BCH Error Decoding for Common Cases

Russ Dill
Arizona State University
russ.dill@asu.edu

Dr. Aviral Shrivastava
Arizona State University
aviral.shrivastava@asu.edu

Dr Hyunok Oh
Hanyang University
hoh@hanyang.ac.kr

Abstract

This paper proposes a new method to optimize a BCH error correction decoder in multi-channel configurations. We break the BCH decoding process into its three basic blocks: syndrome calculation, the error locator polynomial generation, and the roots of the error locator polynomial computation. While an existing multi-channel BCH decoder consists of several single-channel BCH decoders operating in parallel, this paper utilizes a pooled group of shared decoding blocks. By considering the frequency of errors, the proposed pooled group approach requires fewer hardware blocks than in a traditional multi-channel configuration with a negligible impact on performance. Combined with a specialized root finding unit for blocks with only 1 error, our scheme reduces hardware area by 47%–71% and dynamic power by 44%–59% with 2% performance degradation in typical NAND flash systems. With a constant hardware area, the proposed scheme can improve throughput by 3x–5x or NAND flash lifetime by 1.4x–4.5x.

1. Introduction

Error rates in storage and communication channels are increasing [1]. Forward Error Correction (FEC) is a commonly used method to decrease the error rates of those channels[4]. FEC adds redundant information to the message to allow the receiver to correct errors. Bose-Chaudhuri-Hocquenghem (BCH) codes are very commonly used across a wide range of systems [2]. Some of the systems that utilize BCH error correction are; wireless communication links, NAND flash storage, magnetic storage, on-chip cache memories, DRAM memory arrays, and data buses.

Although encoding BCH is fairly straightforward, performing the decoding steps is much more complex [10]. System designers must balance the high complexity of BCH decoders with their overall system requirements [3]. The decoders must provide high throughput, either by running at high clock speeds or by implementing bit-parallel operation. The maximum clock speed of the decoder is limited by the process technology and the complexity of the decoder. Additionally, adding bit-parallel operation increases the area of the decoder and makes it more difficult to achieve high clock speeds. Limited available area for the decoder can also limit the number of errors that can be corrected.

The savings of a more area efficient BCH decoder can be used instead to add bit-parallel operation to improve throughput. Alternatively the decoder could be designed to correct more errors extending the useful life of flash memory or increasing the bit-rate of a communication channel.

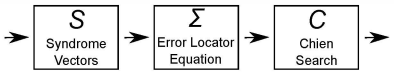

Figure 1. Basic BCH decoder structure

A typical BCH decoder implementation is essentially a 3-stage pipeline as shown in figure 1. The three stages of the pipeline are syndrome calculation, generating the error locator polynomial, and finding the roots of the error locator polynomial [12]. Each pipeline stage operates simultaneously and independently. Data is passed between the stages when the current stage is complete and the next stage is ready to receive the data. This pipelined configuration allows the decoder to operate on 3 codes simultaneously. The first stage, syndrome calculation is similar in fashion to encoding and at similar cost. A simple logic circuit known as a Linear Feedback Shift Register (LFSR) is typically used for syndrome calculation. As LFSRs are used in encoding and syndrome calculation, work has gone to optimize high speed bit-parallel LFSR operation for BCH [25]. Calculating the error locator polynomial, whose roots reveal the locations of errors in the codeword, is performed by successive approximation using the Berlekamp-Massey algorithm. The implementation of the algorithm requires many multipliers and dividers, and consumes a large portion of the decoder. General work into optimized Berlekamp-Massey implementations has been done as well as the sharing of Berlekamp-Massey units between BCH channels. Solving for the roots of the error locator polynomial is typical performed by brute force using an algorithm known as a Chien search [13]. This algorithm searches for roots in the error locator polynomial by evaluating it for each possible error location. The Chien search can be expanded to a bit-parallel architecture. Optimization of this algorithm has been researched heavily, especially in the bit-parallel case due to the large area requirements [22] [26].

Previous works have concentrated on optimizing the stages of single-channel decoders. Much progress has been made on improving the performance and efficiency of individual stages of the BCH decoding process. Although syndrome calculation is the simplest step, it has still received much attention as similar hardware is also used for BCH encoding. As performing operations in a bit-parallel manner can be used to improve performance, Jun et al. [17] have presented work in improving LFSR performance. Additionally, Lee, Yoo, and Park [18] have presented work on improving the syndrome calculation techniques. Generating the error locator polynomial is the most algorithmically complex step of BCH

978-1-4673-8320-2/15/$31.00 ©2015 IEEE

decoding. Compounding the issue, it cannot be modified for bit-parallel operation to improve throughput. Jamro has demonstrated a method of preloading the initial two steps of the algorithm as well as utilizing basis rearrangement to combine two serial steps into one [21]. The final stage of the algorithm is root finding, typically implemented by the Chien search. Kristian has demonstrated the straightforward step to convert the Chien search from a purely serial operation to a bit-parallel operation [14]. As moving to bit-parallel operation quickly increases hardware area, Chen, Yanni, and Parhi have developed a group matching scheme to reduce the hardware complexity in the bit-parallel case [22].

In order to achieve further advances in BCH decoding, we examine the decoding process as a whole and specifically as implemented in multi-channel architectures. A multi-channel BCH decoder is typically designed by putting several single-channel BCH decoders together in parallel. For each set of decoded blocks, only a small fraction of the full error correcting capability is used. For instance, if no error is present in a block, which can be detected during the syndrome calculation, no additional stages are required. If one error is present in a block, the error locator polynomial can be solved directly rather than through a brute force search. For a wide range of error rates, these two cases are very common. Our idea then is to optimize a multi-channel architecture for the common case, rather than the worst case. We use these observations along with the reduced root solver to optimize the stages of the BCH decoder pipeline so that the area requirements are greatly reduced while the optimization incurs a negligible performance degradation. The proposed optimizations reduce power consumption and area requirements greatly. Additionally, by trading our saved area for greater complexity, we can improve throughput and error correcting capability as well.

In this paper, we examine a fixed architecture decoder configured for a representative range of error correction capability. Although the design techniques discussed apply to many uses of the BCH algorithm, in this paper we will be specifically analyzing a decoder for a typical NAND flash controller. This allows us to examine four different possible benefits; increased throughput, lower area, decreased power consumption, and increased flash memory lifetime through greater decoder strength. The base configuration for our decoder is 8 channels, each 4 bits wide running at $200\,\mathrm{MHz}$. This provides a total throughput of $6.4\,\mathrm{Gbit/s}$. We cover decoding strengths of 5 bits, 7 bits, 8 bits, and 10 bits for a data size of $4096\,\mathrm{bit}$ or $512\,\mathrm{B}$. This covers a typical range of error rates. For the design parameters examined in this paper, we achieve an area savings of 47%–71% if we allow a 2% performance degradation. For our test platform, this translates to a dynamic power savings between 44% and 62%. Rather than reducing the area of the optimized design, we can keep the area the same and instead improve performance. Our technique increases throughput by 3x–5x with the same area. Also, we can increase the error correcting capability of the decoder with the same area, which increases the usable life of flash memory. The ageing of flash memory is determined by the number of Program/Erase (P/E) cycles each block has undergone. As the number of P/E cycles increases, the error rate also increases. There is a threshold then where the number of P/E cycles and associated error rate exceeds the error correction capability of the BCH decoder. Although the raw error rate increases rapidly as flash memory ages, our optimized decoder can improve flash lifetime by 1.4x–4.5x.

2. Background and Related Work

2.1 Error Rates

The key component to understanding FEC and the improvements in this paper is understanding error rates.

Information theory tells us that coding systems exist that allow us to use noisy communication channels reliably [23].

BCH is a block based error correction code meaning that it operates on a block of bits at time [11]. It transforms the input data by adding specially calculated redundant check bits to form a codeword. The appropriate code can be selected for a number of bits to be corrected and a chosen block size. Larger block sizes have lower storage overhead, but higher algorithmic complexity.

If the number of errors that occur within the codeword exceeds the capability of the chosen code, an uncorrectable error occurs. This determines the new channel error rate. This rate is calculated by determining the probability that t or fewer errors will occur in a block (where t is the number of errors that can be corrected by the code) and then working backwards to obtain the new bit error rate of the channel. This calculation also accounts for the coding loss, the additional probability that an error will occur in the redundant bits of the codeword.

In order to perform these calculations, the necessary values are the raw channel Bit Error Rate (BER), p, the number of bits in the codeword, n, the error correcting capability of the code, t, and the desired uncorrectable BER. As long as the actual raw channel BER remains at or below the estimated value, the probability of an uncorrectable error occurring will also remain at or below the targeted BER. The most basic calculation is determining that an error free message is received. This is true if every bit in the message is correct [6, p. 168]. We will represent this probability with $P_0(n)$.

$$P_0(n) = (1-p)^n \tag{1}$$

It is straightforward to calculate from eq. 1 the probability that at least one error has occurred, $\neg P_0(n)$.

$$\neg P_0(n) = 1 - P_0(n) \tag{2}$$

$$\neg P_0(n) = 1 - (1-p)^n \tag{3}$$

Moving on from this, we can calculate the probability that exactly m errors occur in a message, $P_{eq}(m,n)$.

$$P_{eq}(m,n) = p^m(1-p)^{n-m}\binom{n}{m} \tag{4}$$

By summing eq. 4 for various values of m, we can calculate the probability that m or fewer errors occur, $P_{le}(m,n)$:

$$P_{le}(m,n) = \sum_{k=0}^{m} P_{eq}(k,n) \tag{5}$$

$$P_{le}(m,n) = \sum_{k=0}^{m}\left[p^k(1-p)^{n-k}\binom{n}{k}\right] \tag{6}$$

We can then use eq. 6 to find the probability that more than m errors occur, $P_{gt}(m,n)$.

$$P_{gt}(m,n) = 1 - P_{le}(m,n) \tag{7}$$

$$P_{gt}(m,n) = 1 - \sum_{k=0}^{m}\left[p^k(1-p)^{n-k}\binom{n}{k}\right] \tag{8}$$

Eq. 8 is important in selecting a BCH code as it shows the probability that a block contains an uncorrectable error. We can then work backwards to find the uncorrectable error rate by plugging the result of eq. 8 into eq. 1 and reversing it.

$$p(t,n)_{uncorr} = 1 - P_{gt}(t,n)^{1/n} \tag{9}$$

Thus given a BER, p, a block size n, and a designed uncorrectable error rate, a sufficient t can be found.

2.2 Flash Memory Lifetime

The push to maximize the storage capacity of NAND flash memory has led to a storage medium that requires extensive error correction in order to be reliable. The primary causes of increasing error rates in flash memory are due to a decreasing process size and an increase in the number of bits stored per cell. Both of these techniques are able to increase storage space well beyond the additional overhead required by Error Correcting Code (ECC).

The properties that lead to high storage densities within flash memory also lead to a lower lifetime. The wearing out of flash memory cells is caused by the high voltages incurred during P/E cycles. These high voltages lead to a deterioration of the tunnel oxide within the cell which then allows leakage. Smaller process geometries have a smaller tunnel oxide layer which wears faster. The smaller process geometries leave less margin for damage that occurs to the cell.

The lifetime of flash memory is rated by the number of P/E cycles it is intended to endure before being retired. Typical P/E lifetimes are rated in thousands of cycles. The targeted lifetime in P/E cycles is chosen as a compromise between durability and ECC requirements. However, by reducing the area and power required by BCH decoding substantially, that compromise can be shifted and the lifetime of the flash memory extended.

The data collected by Cai et al. [5] shows that the relation between P/E cycles and error rates generally follows a polynomial growth. The BER for 3x-nm technology Multi-level Cell (MLC) NAND flash examined in their research closely follows the relation:

$$BER = A * age^2 \tag{10}$$

Where A is a constant specific to a given flash memory. In rearranging the equation to show the relation between age and BER, the constant is eliminated and the following relation is shown:

$$\frac{BER_2}{BER_1} = \left[\frac{age_2}{age_1}\right]^2 \tag{11}$$

So that a doubling of the P/E cycles leads to a quadrupling of the BER. Figure 2 shows the relation between P/E cycles, the BER, and the strength of the BCH code required [5].

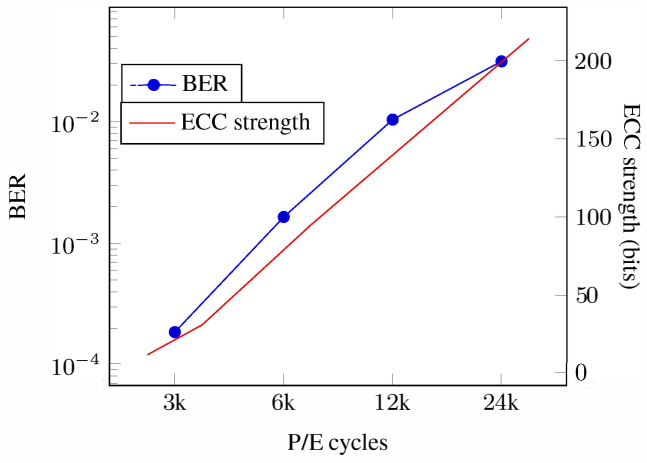

Figure 2. P/E cycles, BER, and ECC strength relation [5]

2.3 BCH Codes

BCH codes are implemented using finite fields. A short overview of finites fields is necessary in understanding both the mechanism of BCH codes and the proposed improvements.

2.3.1 Finite Field Overview

As the name implies, a finite field contains a finite number of elements. Within the set of elements, operations are defined such as addition, subtraction, multiplication, and division. All such operations on field elements result in another field element. Although a wide variety of finite fields can be defined, the use of a binary finite fields makes for a straightforward implementation using digital systems.

A binary finite field is defined by its degree, n, denoted as $GF(2^n)$. The elements of a finite field are created by a generator polynomial. Each element in the field is a successive power of the generator polynomial. Thus the index of the element within the field is known as the power form. For example, for $GF(2^3)$, with a generator polynomial of $x^3 + x + 1$, the field is produced shown in table 1:

Table 1. $x^3 + x + 1$ over $GF(2^3)$

Power form	Polynomial form	Binary representation
0	0	$b000$
x^0	1	$b001$
x^1	x	$b010$
x^2	x^2	$b100$
x^3	$x + 1$	$b011$
x^4	$x^2 + x$	$b110$
x^5	$x^2 + x + 1$	$b111$
x^6	$x^2 + 1$	$b101$

Finite field addition and subtraction is performed by adding or subtracting the polynomial form. Because the order of the field is two (binary field), addition and subtraction are equivalent. In either case, any two equal powers of x cancel out. For example, adding x^2 and $x^2 + x + 1$ produces $x + 1$. This is the equivalent of the logical Exclusive or (XOR) operation.

Finite field multiplication is performed by multiplying the two polynomials together, performing elimination of terms as described above, and then taking the result modulo the generator polynomial. Finite field division is the inverse of finite field multiplication.

When utilizing finite fields for BCH codes, the number of elements in the field is equal to the number of bits within a codeword. For instance, $GF(2^8)$ contains 255 elements (excluding 0). The associated BCH block size would be 255 bits.

In order to make BCH codes easier to work with, only a portion of the codeword is used and the rest of the bits are set to zero. For instance, when using a block size of 16 bytes (128 bits), a BCH code with a block size of 255 bits would be selected. Throughout this paper, codewords are assumed to be constructed in this way.

2.3.2 Finite Field Operations Utilizing LFSR

LFSRs are commonly used for finite field operations. The basic operation of a LFSR allows one to transform a finite field element to the next or previous element within the field. This is equivalent to multiplying or dividing by x^1. Thus repeated operation can multiply or divide by any power of x.

A LFSR consists of a set of registers interconnected in a ring configuration. Between each register there can be an XOR gate. The XOR gate combines the value of the previous register with feedback from the highest register. An example LFSR is shown in

figure 3. The configuration shown can be used to produce the finite field shown in table 1. This is because the connections match the binary representation of the generator polynomial. In this configuration, the LFSR will cycle through each element of the field in order.

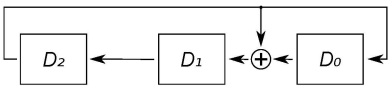

Figure 3. Example LFSR

LFSRs are commonly used for BCH operations, either in their default form, or in a slightly modified form that allows other operations, such as determining the remainder of a division [19].

2.3.3 Encoding

BCH encoding is performed by dividing the input data by a specially formed polynomial. This is performed utilizing a modified LFSR that accepts a bit of input data per clock cycle. At the end of the operation, the LFSR contains the remainder of the operation which is the redundant code bits [20].

2.3.4 Decoding

The decoding process is broken into three stages which operate independantly. The input codeword is passed into the first stage and error locations are generated by the final stage. Figure 4 shows the hardware stages of the decoding process. In the figure, the red squares within the codeword represent error locations.

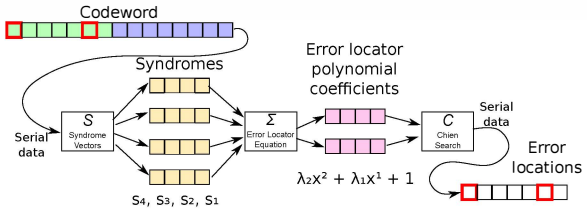

Figure 4. BCH decoding process

2.3.5 Decoding - Syndrome Computation

The syndromes are a set of values that once computed, depend only on the error locations within the message, and not on the message itself. The number of syndromes is twice the number of errors that the BCH code can correct, t. The syndromes are generated by dividing the codeword by a set of minimal polynomials producing a set of remainders. Because of relations between the minimal polynomials, many syndrome elements can be easily derived from the other elements, reducing the amount of computation required. A useful property of the syndromes is that if all calculated syndromes are zero, then no errors exist in the received message.

Syndrome computation operates on one input bit at a time which limits the overall bandwidth of the decoder to the clock rate of the syndrome units. However, syndrome calculation can be modified to perform bit-parallel operations, greatly increasing the throughput of the syndrome calculation stage at the cost of increased area and power.

2.3.6 Decoding - Error Locator Polynomial Generation

The error locator polynomial is defined such that its roots give the locations of the errors within the message. The number of roots, or degree, of the error locator polynomial indicates the number of errors within the message. The second stage of the BCH decoding process is to generate the error locator polynomial from the set of syndromes.

The Berlekamp-Massey algorithm was developed to generate the error locator polynomial from a set of syndromes. It is an iterative algorithm which calculates a discrepancy at each stage, refining the approximation. This process requires several finite field multiplications, divisions, and additions per cycle of the algorithm which contributes to the overall complexity of the decoder.

2.3.7 Decoding - Root Finding

To find error locations, roots of the error locator polynomial must be found. Since the degree of polynomial can be as large as t, a brute force algorithm is used for hardware BCH implementations. An optimized algorithm used for this brute force search has been developed and is known as a Chien search. To implement the Chien search, a set of registers is loaded with the coefficients of the error locator polynomial. During each cycle of the Chien search, each register is multiplied by x^n, where n is the degree of x associated with the given coefficient. At the end of each cycle, all registers are summed. If the sum of all the registers is zero, then a root has been located. The cycle number indicates the index within the block of the error location.

The order of the Chien output can be made to match the order of the input message. Thus the output of the BCH decoder is a set of locations within the message that must be toggled to correct received errors.

2.4 Current Methods of Improving Performance

Although increasing clock rate leads directly to an increase in throughput, there is a limit due to the complexity involved in the decoder. There are two other methods of increasing the throughput, implementing bit-parallel operation in the syndrome calculation and root finding, and implementing multiple BCH decoders in a system operating in parallel.

Both the input and output of the BCH decoder handle data serially, one bit per clock cycle. The logical result of multiple clock cycles can be combined allowing the input and output to operate on multiple bits in parallel. Bit-parallel operation is a straightforward implementation and typically requires few modifications to an overall system to implement. However, as bit-parallel operation increases the complexity of the decoder, it decreases the achievable clock rate and thus has limits. Additionally, bit-parallel operation cannot be applied to generating the error locator polynomial, and thus the overall throughput of the system will come to be limited by this step.

Implementing multiple BCH channels bypasses these problems as it is simply a duplication of the BCH engine. Multiple channels require modification of the overall system to implement and can be made in two primary situations.

The first is the case of a multi-channel architecture. For example, a system that has multiple data channels connected to flash memory [8].

The second is to interleave the BCH code. Interleaving not only leads to increased throughput, but also offers error correction advantages in certain types of channels [9]. This is because in many types of channels, errors tend to occur in bursts. With interleaved operation the burst is broken up across many codewords, decreasing the probability that a single burst will overwhelm the error capability of the chosen BCH code [7].

Both methods of multi-channel operation scale each property of the system (throughput, area, power) in a purely linear fashion.

2.5 Current Methods of Improving Efficiency

Improving the efficiency of each stage of decoding can lead to lower area requirements, lower power consumption, and increased clock speeds leading to higher throughput. As such, many ideas

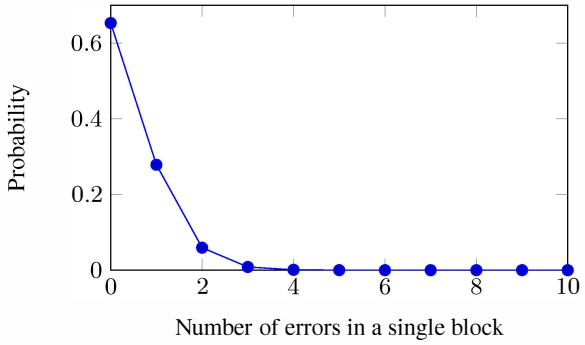

Figure 5. Probabilities of errors at BER of 1×10^{-4}

have been put forth to improve the efficiency of each BCH decoding stage.

For instance, it has been shown that a relation exists between many of the syndromes [24, p. 152]. This makes it possible to only calculate a limited set of syndromes, and then apply the relations to expand them into the full set of syndromes. This decreases the overall area and power requirements of the decoder.

Additionally, it has been shown that there are multiple methods of finding each syndrome element [27]. For a given element, it can be shown which method is the most efficient. This information can then be used to calculate each syndrome in the most efficient way possible. This not only decreases the overall area and power requirements of the decoder, but because it decreases complexity, can also increase clock speeds and throughput.

Work has also gone into decreasing the complexity of bit-parallel LFSRs. This work can be and has been applied to bit-parallel syndrome calculation [25].

As the step of generating the error locator polynomial can limit the overall throughput of the decoder, improving its efficiency, increasing the achievable clock rate, and decreasing the overall number of clock cycles required is important. General optimizations to finite field operations, such as more efficient multipliers and dividers, can be applied to generating the error locator polynomial.

Jamro has shown how linking multipliers which operate on different bases can lead to a reducing in the number of clock cycles required [21]. This is done by linking a serial multiplier that takes parallel input and produces serial output with a multiplier that takes serial input and produces parallel output. However, as these two multipliers operate on a different bases, an efficient basis conversion circuit linking the two multipliers is shown. Additionally, Jamro shows how the first two rounds of the algorithm can be skipped by pre-calculating the necessary state of the registers. Both of these optimizations reduce the latency of generating the error locator polynomial. By reducing the latency, this allows the decoder to run at a higher overall throughput.

The Chien search requires a number of multipliers equal to the number of coefficients in the error locator polynomial [22]. Additionally, bit-parallel operation requires a duplication of this set of multipliers for each output bit as well as a multiplier to load each coefficient with the appropriate value.

3. Main Observations

In order to push the uncorrectable error rate very low, BCH decoders are very oversized compared to the number of errors they typically correct. The common case is for only a fraction of the decoder to be used. This is shown clearly in figure 5.

This observation alone does not allow us any improvement because at any time the full decoder may be required. We instead observe that on average only a small percentage of the decoder is required and then apply that observation to a multi-channel decoder. By applying our observation to a multi-channel decoder, we can include at least one full BCH decoder. The remainder of the decoding hardware can be reduced decoders of some kind. These reduced decoders could reduce our overall hardware requirements greatly.

To route data properly, we need to consider how many errors a block has. Assume that there is no error. All syndromes are evaluated to zero and the block needs no further processing.

To calculate the number of errors beyond zero, we must find the error locator polynomial. Any reduction in the complexity of the decoder beyond zero errors must then be in the root search. The case of only one error is a very common case and a good target to optimize for. The optimization here is fairly straightforward as the error locator polynomial will only be one degree in this case. Rather than a brute force search, the root can be found algebraically.

The trade-off with such a system is that there is a possibility that insufficient resources will be available to decode a certain set of blocks. For example, if 5 of the 8 blocks contain errors but only 4 error locator units are present. If this occurs, decoding will be delayed until resources are available and performance will be degraded. Fortunately, it is fairly straightforward to calculate this performance drop and thus intelligently trade-off a small drop in performance for a large reduction in area and power requirements.

4. Our Approach

4.1 Architecture

The basic design of a BCH decoder is broken down into three pipeline stages. For our multi-channel architecture, we implement those stages as stations fed by round robin arbitrators. The arbitrators collect data from each stage and then passes it to the next. The general layout of the decoder is shown in figure 6. In the example configuration, there are 3 error polynomial generator units (Σ), one traditional Chien solver (C) and two reduced root solvers (r).

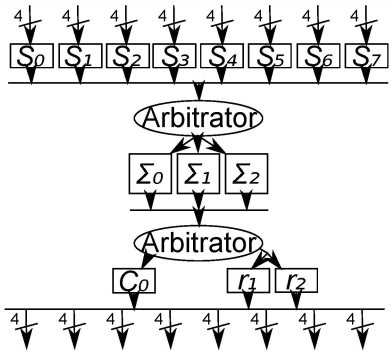

Figure 6. An example of the proposed BCH decoder

The overall architecture can be configured for a given number of channels, error locator polynomial generators, traditional Chien search units, and reduced root solver units.

4.1.1 Syndromes

For every channel, the syndromes must be computed. This means that the number of syndrome units will be equal to the number of channels. We fix each syndrome unit to a channel and each unit contains a bit counter. The counter will be used to track how many bits the unit has received and if the syndrome is ready.

On the input side, the syndrome unit contains two control signals. An input to indicate that it should start accepting syndrome data, and an output that acknowledges that signal. If the unit is busy

or contains processed syndrome data, it will not acknowledge the start signal.

On the output side, the syndrome unit contains an additional two control signals. One signal indicates that the syndrome unit contains processed syndrome data. The other control signal is an input that clears this state and allows the unit to accept new data.

4.1.2 Syndrome/Error Locator Polynomial Interconnect

This interconnect passes data from the channel syndrome units to the pool of error locator polynomial generators. The unit primarily consists of a register to hold the syndromes, an index to the current syndrome input unit, and an index to the current error locator polynomial unit. Both indexes operate in a purely round robin fashion. The unit also contains a circuitry to check its currently stored syndrome against zero. It determines if it is necessary to pass the syndrome data to the error locator polynomial unit or if it can be skipped.

The general operation is to wait on the currently indexed syndrome unit. When a syndrome is ready, it accepts the syndrome and stores it in its syndrome register. It also stores the index to associate the data with a channel. It then waits for the syndrome to be compared against zero. If the check indicates no errors are present, it sets a flag indicating that the current channel output should skip root finding for the next data set.

If the check indicates errors are present, it waits for the next error locator polynomial generator unit to become ready. When ready, it passes its syndromes to that unit and sets the start bit for that unit. It also passes the currently stored channel number so that the error locator polynomial will be associated with the correct channel.

4.1.3 Error Locator Polynomial Generator

If any error exists within the codeword, we must find the error locator polynomial. The control signals on this unit are similar to the control signals on the syndrome unit. A start and start acknowledge signal on the input, and a signal to indicate done state and a signal to clear the done state on the output.

The output of the error locator polynomial generator unit includes the error locator polynomial and also the number of errors detected within the codeword. The only configuration available for the error locator polynomial are the BCH code parameters.

4.1.4 Error Locator Polynomial/Root Solver Interconnect

This interconnect is similar to the syndrome interconnect except that it must serve two possible pools. The first destination pool consists of traditional Chien root solvers and the second destination pool consists of reduced root solvers. When the currently selected error locator polynomial is ready, the interconnect stores the error locator polynomial, the error count, and the associated channel number.

The interconnect must then determine based on the error count which pool to serve. It keeps two separate indexing counters, one for each pool. If the error count is 1 then the reduced root solver pool is used, otherwise the traditional Chien pool is used.

When the appropriate root solver is ready, the interconnect signals it to start and passes the error locator polynomial along with the associated channel number.

4.1.5 Traditional Chien Root Solver

The traditional Chien root solver units consist of a set of coefficient registers. Each register is wide enough to contain a finite field element from the given BCH configuration. The number of registers required is equal to the maximum number of errors that the code can correct. The registers are each multiplied by the appropriate degree of x each cycle and each cycle all registers are summed together. If the sum is zero, then an error has been located. This operation is duplicated for bit-parallel operation, with the number of bits shared per register being configurable in order to meet timing. Additionally, the summing operation provides an opportunity for a configurable amount of pipelining.

The unit contains a start signal that is used to load new values in the coefficient registers, starting the algorithm. Due to the pipelined nature of the summation operation, an output signal is provided that indicates that the first bit (or set of bits) of errors is being output on the current cycle.

The glue logic surrounding the root solver contains a multiplexor that connects to the busy signal of the output stages. The output stage then counts the number of cycles necessary for the algorithm to complete.

4.1.6 Reduced Root Solver

The reduced root solver can be used to find the error location for codewords with a single bit error. It offers large advantages over the traditional Chien search since it only requires a single register. It also is more efficient in the multi-bit case as for each bit, since the register is compared against a constant.

If only one error exists in a codeword, the error locator polynomial is of degree 1 and of the form:

$$Ax + B = 0 \tag{12}$$

Which can be solved in a single step as:

$$x = -B/A \tag{13}$$

Because of the algorithm we use to find our error locator polynomial, B is always 1. Additionally, negation is a null operation within finite fields. This reduces the equation further to the form:

$$x = 1/A \tag{14}$$

Although implementing an inverter would produce the value of x in a single cycle, the value would be of little use on its own. This is because the value is in the standard basis for the finite field and not the power form. The power form would give us a direct integer index to the location of the error. The binary representation of the sequencing of the standard basis (polynomial form) can be seen in table 1.

Converting from the power form to the standard basis is an algorithmically complex operation. It is generally on the order of $O(N)$ where N in the number of elements in the field. Rather than attempt to convert from the power form to the standard basis, we make two observations.

Our first observation is that we need to cycle through each bit in the codeword in order to output error locations regardless of how our solver functions. Our second observation is summed up by the following re-arrangement:

$$Ax = 1 \tag{15}$$

If we load a register with A and multiply it repeatedly by x^1, it will eventually reach the value of 1. Once it has we have multiplied A by the correct power of x and found the root. Because we are only multiplying by x^1 per cycle we can use a LFSR instead of a multiplier.

To start, we load the LFSR with the value of A. Then during each cycle, we advance the LFSR and compare the value with 1. If they match we have found the location of the root.

Expanding this to support multiple bits scales very well. We advance the LFSR a number of cycles equal to the number of bits instead of just once. For each output bit, we compare the value in the LFSR with the next value in the finite field starting with 1 for the first bit.

4.1.7 Output Units

The output units multiplex the data from the root solvers and output it from the decoder. Each channel has an associated output unit. The output units provide the data indicating which bits are in error as well as a signal to indicate the start of a new block. Within each output unit is a counter to keep track of when the output for the given block is complete and the next block can be processed.

The output units are driven by two flags. One flag indicates that the output unit should expect data from a root solver, the other flag indicates that the output unit should output one block's worth of error free data. Whenever the output unit completes its current block, it examines these flags to determine what it should output next.

Whenever the flag indicating that data from a root solver should be processed, the associated index of that solver is stored as well. This allows the output unit to assign its multiplexor to accept data from the appropriate solver.

4.2 Determining the Number of Units

Part of the design is to select the appropriate number of each unit type. The number of units included in a given design is determined by the expected error rate and the acceptable miss rate. The miss rate indicates the likelyhood that within any given set of blocks, there would be insufficient hardware to process the data. In this case the effected input channel is stalled and the decoding of that block is deferred until hardware is available. The overall throughput of the system is reduced by the miss rate.

We need to decide the number of units in two stages. The first stage is the error locator polynomial generator units. Units are only required for blocks with one or more errors. Therefore the number of units is chosen based on the probability that more than m blocks contain one or more errors. We start by using eq. 2 to determine the probability that a single block contains one or more errors. Then we plug this probability into eq. 8 and choose the message size n to be equal to the number of channels. By evaluating this equation for different values of m, we can find the number of blocks required to be below the miss rate probability.

The result of evaluating this equation for the chosen set of BCH parameters and an acceptable miss rate of 2% is shown in figure 7.

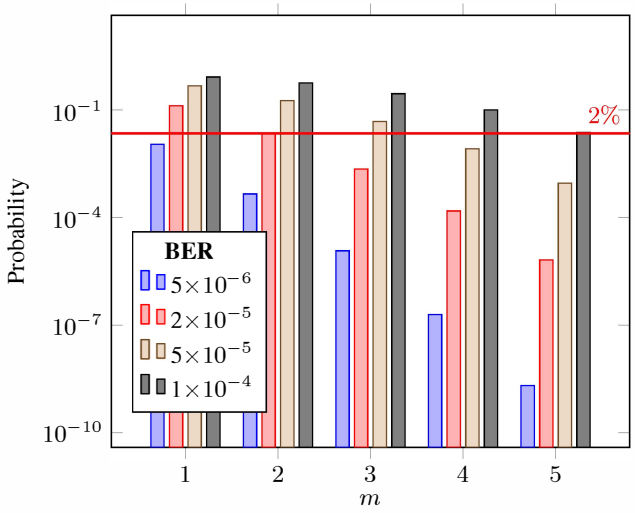

Figure 7. Probability that more than m blocks contain at least one error where $n = 8$

Figure 7 shows that for a BER of 5×10^{-6}, only 1 unit is required with a 2% miss rate. For a BER of 1×10^{-4}, 5 units are required.

The next step determines the number of traditional Chien search units required. This is calculated similarly to the above, but we examine the probability that more than one error occurs since our reduced root solver can only handle one error. We first use eq. 8 to find the probability that more than one error occurs within a single block. And then we use eq. 8 again, but this time with the message size set to the number of channels and p set to the value found above.

The result of evaluating this equation across multiple values of m is shown in figure 8.

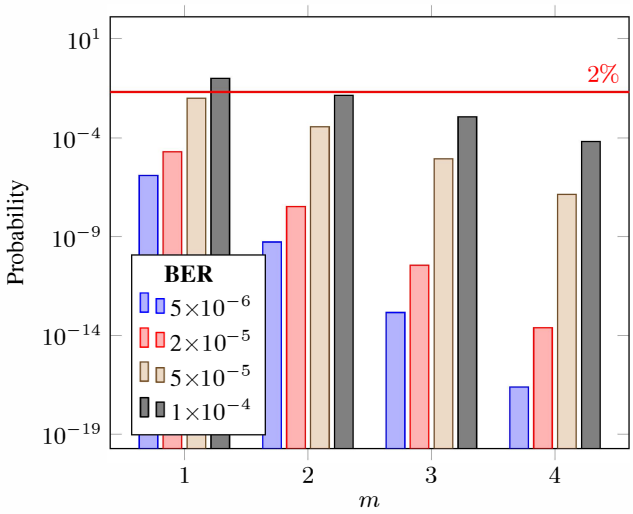

Figure 8. Probability that more than m blocks contain more than one error where $n = 8$

Figure 8 shows that for a BER of 1×10^{-4}, 2 units are required. For all other examined error rates only 1 unit is required. The remaining units are filled in with reduced root solvers. Note that for any decoder at least one traditional Chien solver is required.

A miss rate of 2% is chosen for our experiments as it is a very small performance penalty, but still large enough for smaller unit counts to be used. In order to demonstrate the variability of units required for a given miss rate, the BER of 2×10^{-4} is examined. This BER provides a wide range of required units across a set of given miss rates.

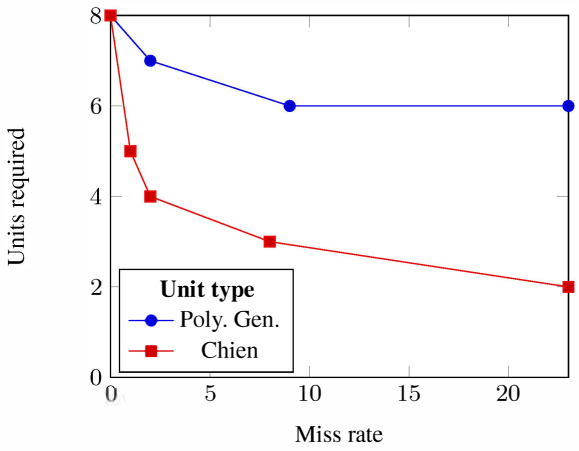

Figure 9. Units required for BER of 2×10^{-4}

As shown in Figure 9, the gain seen for a given miss rate falls off quickly beyond 2%. Although 2% was chosen for the exper-

iments in this paper, the additional gains achieved through much higher miss rates may still be desirable on extremely constrained designeds.

5. Experiments

5.1 Setup

In order to test our ideas and approach, we have implemented them on a Field Programmable Gate Array (FPGA) in Verilog. A Xilinx Virtex-6 FPGA has been chosen as a target as it has sufficient logic resources and Input/Outputs (IOs) for implementing the necessary experiments.

Our Verilog code is written to be configured through Verilog parameters. This allows the properties of the decoder to be configured at compile time. The build tools can then compile and verify a variety of configurations in a batch form without modification to the codebase.

Validation of the design is performed with a set of testbenches. These verify the correctness of the compiled code and algorithms. The testbenches operate by generating a stream of random input data as well as random bit errors. Because the natural probability of the maximum number of correctable errors occurring is extremely low, each possible number of errors is selected equally. This allows the code to be fully exercised. The input data is fed to a BCH encoder and then bits are flipped in accordance with the generated error locations. The modified data is then passed to the BCH decoder and the error locations output is compared with the true error location data.

The area of a given design is calculated by implementing the design fully. All inputs and outputs of the design are assigned registers as would be done in a system design to meet IO timing. As the configurability of the design leads to a wide range of IO configurations, the tool is permitted to automatically assign IO locations. The comparative area of design is then measured by FPGA slice usage.

Power estimation is performed using the Xilinx XPower Analyzer. Since the static power consumption of an FPGA does not vary significantly based on logic usage, dynamic power consumption is compared.

In order to ensure a fair comparison, all designs are constrained to run at at least 200 MHz. This ensures that complex designs will pay an area penalty as the tool will duplicate registers to meet timing.

5.2 Baseline Configuration

Our baseline configuration is an 8 channel decoder. Not only do many systems contain a similar number of channels, but it also allows us to fully demonstrate the advantages of our approach.

Each channel is 4 bits wide. Most flash memory systems operate in an 8 bit wide configuration, but a 4 bit wide configuration was chosen for two reasons. First to allow the design to have headroom for demonstrating the increase in throughput possible in the optimized design. Second, many decoders operate at a higher clock rate than the data bus. For a decoder operating at double the clock rate of an 8 bit data bus, 4 bit wide operation would be required.

The baseline decoder operates on 4096 bit, or 512 B blocks. This is a typical block size for the error rates examined in this paper [15][16]. Similar results should be obtainable across a wide range of possible block sizes.

Flash manufacturers typically do not publish BER values for released flash memory. They instead publish the error correction strength required to reduce the error rate below an acceptable threshold, typically 1×10^{-15}[5]. Knowing the error correction strength required, the block size, and the targeting uncorrectable error rate, we can work backwards to estimate the associated BER. The values chosen are shown in the table 2.

Table 2. Targeted ECC Range

Strength (errors)	Estimated BER	Bits of ECC required
5	5×10^{-6}	65
7	2×10^{-5}	91
8	5×10^{-5}	104
10	1×10^{-4}	130

5.3 Area Optimized BCH Decoder

Our area optimized BCH decoder reduces the hardware area while it impacts performance only 2%. The reduced number of units required is shown in the table 3. Although 8 syndrome calculation blocks are always required, the number of error locators and traditional Chien search blocks decreases as BER decreases to meet the given error rate with 2% miss rate.

Table 3. Hardware Units Required

BER	Syndrome	Error Locator	Trad. Chien	Reduc. Root
5×10^{-6}	8	1	1	0
2×10^{-5}	8	3	1	2
5×10^{-5}	8	4	1	3
1×10^{-4}	8	5	2	3

The area is then compared with the baseline decoder and the results are shown in Figure 10. Note that the area includes all hardware components such as arbitrators to build the BCH decoders which are not required in the baseline implementation. By optimizing the number of units and utilizing the reduced root solver we reduce required area by 47%–71% compared to the baseline implementation.

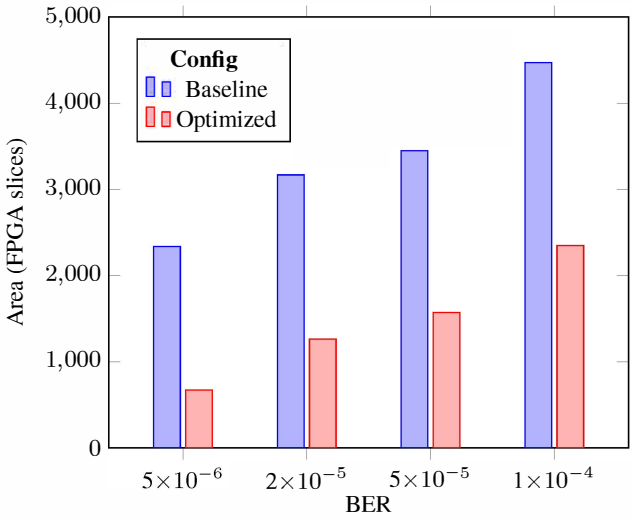

Figure 10. Area saving results

The smaller area of the decoder also translates to dynamic power savings. By profiling the designs we can estimate the power consumed by each design. The results are shown in figure 11. This equates to a 44%–59% reduction in dynamic power requirements.

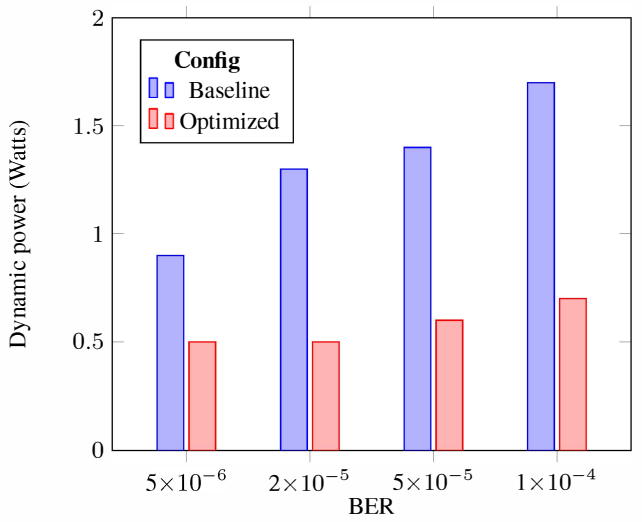

Figure 11. Power saving results

5.4 Throughput Optimized BCH Decoder

While the proposed area optimized BCH decoder sacrifices a small amount of performance to reduce the required hardware area, it is possible to devise a throughput optimized BCH decoder while holding area constant to improve the performance. The optimization is achieved by increasing the bit-parallel configuration parameter until a maximum throughput is found at the same area cost as the baseline configuration.

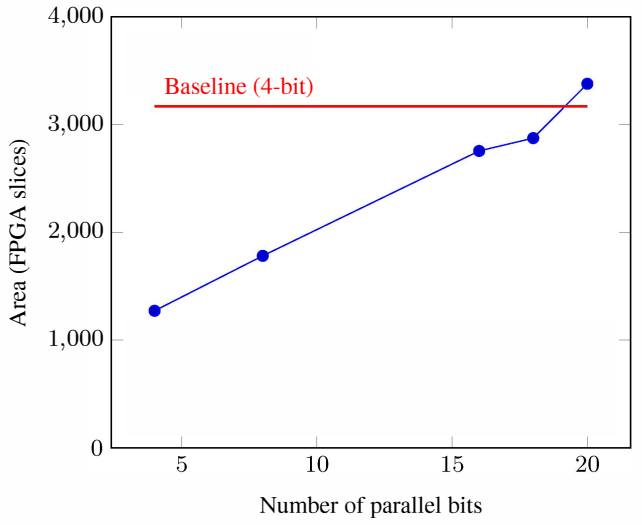

Figure 12. Requirements of 2×10^{-5} design

Figure 12 shows the process as applied to the 2×10^{-5} BER configuration. The area consumed by the baseline unoptimized design is shown by the red line. The discontinuity in results is due to an additional level of hardware duplication in the Chien search when moving to a 20 bit wide unit to meet timing.

While the unoptimized design consumes an area of 3168 slices, our optimized design consumes an area of only 1272 slices. Both designs accept 4 bits per cycle in the syndrome calculation stage, and output 4 bits per cycle in their output stage. We then implement the optimized design at 8 bits, 16 bits, 18 bits, and 20 bits per cycle. These designs increase throughput by operating on more input and output bits per clock cycle. We can see that the optimized design operating at 18 bits per cycle only consumes 2874 slices, which is less than the unoptimized design operating at only 4 bits per cycle.

Thus it is possible to implement an 18-bit design within the same area, leading to a 4.5x improvement in performance. Note that there is 2% performance degradation due to the miss rate, which is negligible compared with the performance gain of 450%. Similar improvements in performance are possible with our other configurations and are shown in figure 13. The amount of performance improvement is related to the area savings provided by the optimized decoder.

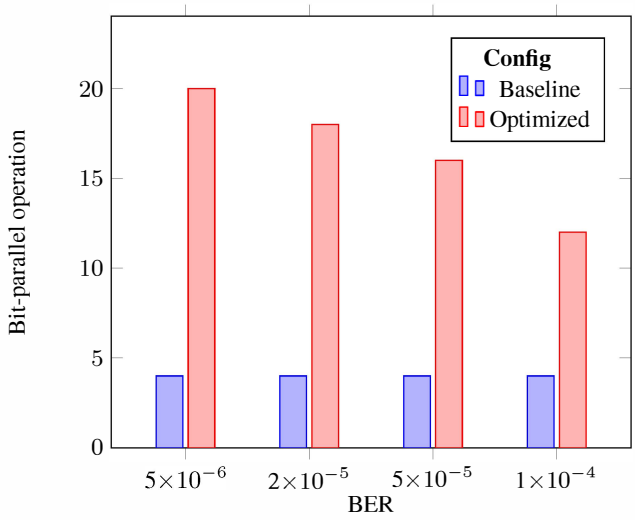

Figure 13. Throughput optimization results

5.5 Flash Lifetime Optimized Design

Similarly, the area reduction can be utilized to increase the lifetime by providing higher error correction strength. To provide stronger error correction, a larger hardware area is required. The area reduction in our approach is utilized to provide greater error correction strength in a smaller area. For an 8 channel unit and a 2% targeted miss rate, the hardware area requirement in our approach for a BER of 1×10^{-4} becomes similar to the area in the baseline approach for a BER of 5×10^{-6}. Table 4 shows the units required in our approach for different given BERs.

Table 4. Hardware Units Required

BER	Syndrome	Error Locator	Trad. Chien	Reduc. Root
1.2×10^{-4}	8	6	3	3
1.5×10^{-4}	8	7	3	4
2.0×10^{-4}	8	7	4	3

Table 5 compares the error correction capability between the baseline approach and the proposed optimization with a given hardware area constraint. For instance, for the hardware area with which the baseline approach can handle a BER of 5×10^{-6}, the proposed approach can handle a BER of 1×10^{-4}. Note that in the table, our approach requires no larger hardware area than the baseline approach. In addition to increased error correction capability, our implementation includes additional hardware units to meet the 2% miss rate. Therefore, our approach can correct more errors than the baseline approach without sacrificing performance, hardware area, and power consumption.

Table 5. Error correction capability

Original BER	Original t	Optimized BER	Optimized t
5×10^{-6}	5	1.0×10^{-4}	10
2×10^{-5}	7	1.2×10^{-4}	11
5×10^{-5}	8	1.5×10^{-4}	12
1×10^{-4}	10	2.0×10^{-4}	13

Equation 11 shows the relation between BER and ageing. Since the proposed scheme can correct more errors, allowing a decoder targeted for a higher BER, the lifetime of the same NAND flash memory is prolonged compared with the baseline implementation. Figure 14 shows the lifetime improvement over the baseline BCH decoder. As a BER decreases, more hardware reduction is achievable and more errors can be corrected by utilizing the reduced area. The flash lifetime is extended by 1.4x–4.5x.

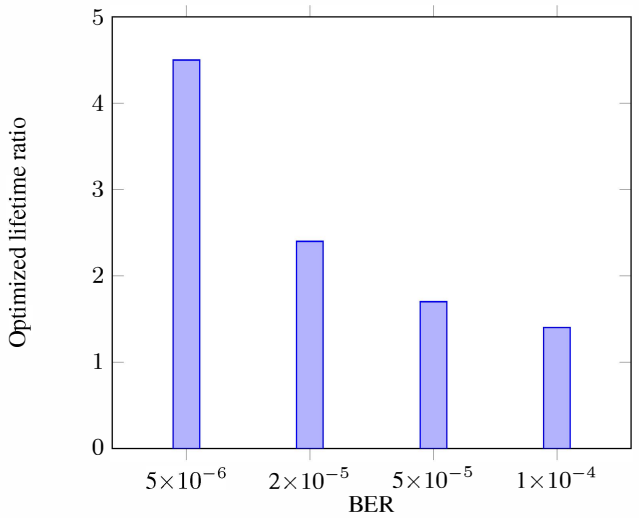

Figure 14. Improved lifetime

6. Conclusion

This paper proposes new multi-channel BCH error correction decoder optimization techniques to reduce the hardware area requirement by considering a common error case. The proposed scheme utilizes a pooled group of shared decoding blocks. Compared with a traditional multi-channel implementation, it reduces the hardware area by 47%–71%. The area reduction also saves the dynamic power consumption by 44%–59%. In our approach, if the reduced hardware area is utilized to increase the performance, the throughput is improved by 3x–5x and the lifetime of NAND flash increases by 1.4x–4.5x if it is utilized to correct more errors.

References

[1] Luyi, Sui, Fu Jinyi, and Yang Xiaohua. "Forward error correction." Computational and Information Sciences (ICCIS), 2012 Fourth International Conference on. IEEE, 2012.

[2] Sun, Fei, Ken Rose, and Tong Zhang. "On the use of strong BCH codes for improving multilevel NAND flash memory storage capacity." IEEE Workshop on Signal Processing Systems (SiPS): Design and Implementation. 2006.

[3] Strukov, Dmitri. "The area and latency tradeoffs of binary bit-parallel BCH decoders for prospective nanoelectronic memories." Signals, Systems and Computers, 2006. ACSSC'06. Fortieth Asilomar Conference on. IEEE, 2006.

[4] Rate, Switch. "Forward error correction schemes for digital communications." (1983).

[5] Cai, Yu, et al. "Error patterns in MLC NAND flash memory: Measurement, characterization, and analysis." Design, Automation & Test in Europe Conference & Exhibition, 2012. IEEE, 2012.

[6] Houghton, A., ed. Error coding for engineers. Springer Science & Business Media, 2001.

[7] Shi, Yun Q., et al. "Interleaving for combating bursts of errors." Circuits and Systems Magazine, IEEE 4.1 (2004): 29-42.

[8] Abraham, Michael. "NAND flash trends for SSD/Enterprise." Flash Memory Summit (2010).

[9] Lee, Kihoon, et al. "100GB/S two-iteration concatenated BCH decoder architecture for optical communications." Signal Processing Systems (SIPS), 2010 IEEE Workshop on. IEEE, 2010.

[10] Zambelli, Cristian, et al. "A cross-layer approach for new reliability-performance trade-offs in MLC NAND flash memories." Proceedings of the Conference on Design, Automation and Test in Europe, 2012.

[11] Bose, Raj Chandra, and Dwijendra K. Ray-Chaudhuri. "On a class of error correcting binary group codes." Information and control 3.1 (1960): 68-79.

[12] Hong, Jonathan, and Martin Vetterli. "Simple algorithms for BCH decoding." Comm., IEEE Transactions on 43.8 (1995): 2324-2333.

[13] Litwin, Louis. "Error control coding in digital communications systems." RF Design, July (2001).

[14] Kristian, Hans, et al. "Ultra-fast-scalable BCH decoder with efficient-Extended Fast Chien Search." Computer Science and Information Technology (ICCSIT), 2010 3rd IEEE International Conference on. Vol. 4. IEEE, 2010.

[15] Cooke, Jim, Berrett, B., Schulthies, V. "101: An Introduction to NAND Flash and How to Design It in to Your Next Product." Micron (2006): 1-28.

[16] Cooke, Jim. "NAND 201: An Update on the Continued Evolution of NAND Flash" Micron (2011).

[17] Jun, Zhang, et al. "Optimized design for high-speed parallel BCH encoder." VLSI Design and Video Technology, 2005. Proceedings of 2005 IEEE International Workshop on. IEEE, 2005.

[18] Lee, Youngjoo, Hoyoung Yoo, and In-Cheol Park. "Small-area parallel syndrome calculation for strong BCH decoding." Acoustics, Speech and Signal Processing (ICASSP), 2012 IEEE International Conference on. IEEE, 2012.

[19] Saluja, Kevval K. "Linear Feedback Shift Registers Theory and Applications." Department of Electrical and Computer Engineering, University of Wisconsin-Madison (1987): 4-14.

[20] Lee, Je-Hoon, et al. "Implementation of Parallel BCH Encoder Employing Tree-Type Systolic Array Architecture." (2013).

[21] Jamro, Ernest. "The design of a VHDL based synthesis tool for BCH codecs." The university of Huddersfiel (1997).

[22] Chen, Yanni, and Keshab K. Parhi. "Small area parallel Chien search architectures for long BCH codes." IEEE Transactions on Very Large Scale Integration (VLSI) Systems 12.5 (2004): 545-549.

[23] Shannon, C. E. "A Mathematical Theory of Communication." Bell System Technical Journal, July 1948, p.623

[24] Lin, Shu and Costello, Daniel J. "Error control coding: fundamentals and applications." Pearson-Prentice Hall Upper Saddle River, 1983

[25] Lowy, Menahem. "Parallel implementation of LFSRs for low power applications." Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on 43.6 (1996): 458-466.

[26] Hu, Qingsheng, et al. "Low complexity parallel Chien search architecture for RS decoder." Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium on. IEEE, 2005.

[27] Cho, Junho, and Wonyong Sung. "Efficient software-based encoding and decoding of BCH codes." Computers, IEEE Transactions on 58.7 (2009): 878-889.