A New Error Correction Code

Amir Shahab Shahmiri, Sobhan Naderi Parizi, Mohammad Kazem Akbari
Computer Engineering & Information Technology Department
Amirkabir University of Technology,
Tehran, Iran
amir@shahmiri.ir, {snaderi, akbarif}@aut.ac.ir

Abstract—**There are some design procedures that simplify fault diagnosis or detection in which faults can be automatically detected and/or corrected by use of coded inputs. In general, codes are commonly classified in terms of their ability to detect or correct classes of errors that affect some fixed number of bits in a word. Many codes have been developed that can be used in the design of self-checking circuits. Type of codes may vary depending on the type of circuits. For data-transmission busses, a parity-check code may be adequate, for other types of functions, however, we may wish to use a code by which the check bits of the result can be determined from the check bits of the operands. In this study, we developed a new Error Detection and Correction Code (ED/CC), called "Persec code", which proved mathematically to be better in compare with other candidates and also adaptive to changing environments. Theoretically, this code is able to detect several errors, and correct more than one error of data-packet as well. This paper successfully demonstrates 1-error correcting scenario, via simulation and validation processes.**

I. Introduction

During transmission of information via communication networks data may get corrupted due to physical/logical faults which would bring the whole system down to destructive failures. So, every communication system has to be facilitated with testing and fault tolerance equipments, to provide safe and sound communication streamlines.

So far, many error detection and error correction codes, for different purposes, have been developed. To name some, Parity codes, Burger codes and Checksums[1-3] for error detection, Cyclic Redundancy codes[4,5], Hamming codes[6], Residue codes[1-3,7], Nordstorm-Robinson codes[8] and Turbo codes[9] for error correction, and BCH codes[10-12] and modified Residue codes[13,14] for multi error correction were developed. These codes may perform well in some cases, but not in all conditions and environments. However, due to steady increase of size, speed and complexity of data transmission the total efficiency has been reduced. Therefore, vital need for creation of new methods and revising the old techniques is commonly sensed [15].

Generally, some of the faults are due to magnetic fields, electrical influences and climate impacts, such as thunders, hurricane, solar rays and etc. They can appear in both internal (e.g., inter node computer communications) and external communications (e.g., satellite communications, digital telecommunications or wireless networks). Traditionally, memories employ Single-Error Correcting and Double-Error Detecting (SEC-DED) methods [16]. But in telecommunications with large data-packets, systems need error correcting methods along with multi-error detecting techniques. Hence the task of every receiver system is to check errors and then fixing the problem by requesting for re-transmission, correction or using other means.

In this regard, many methods have been developed by different designers, which were good only in specific conditions and environments. However, an ED/CC is a mathematical function [2] that usually is implemented by hardware devices. One of the most popular error-detecting techniques is Parity code. Parity code is fast and efficient, because uses only one extra-bit (check bit) per data-packet (usually a byte) for detecting odd number of errors in the packet [17]. Hamming code, another well-known error-correcting code, makes c extra check-bits per data-packet out of k information bits, where $2^c = c + k + 1$ [3,6]. The cost of these check-bits is very low in large data-packets but require more time for the correction processes.

Since these codes need some extra-bits for the detection/correction processes which impose more time overhead, so designers depending on different conditions, nature and behavior of transmitter-receiver systems need to choose the best methods to increase the performance of the checking system.

This paper is articulated as follows: In section 2 terminology used for Persec code to understand this technique better is discussed. Persec code's algorithm and its characteristics are clarified in section 3. The simulation results are analyzed in section 4, and finally presents the conclusion and future works in section V and VI.

II. Terminology

During the Achamenian dynasty, the Persian empire had designed a special task force called "Sepah Javidan" (also called immortal/eternal army) which was comprised of a fixed number of ten thousands warriors. In times, within the battles some combatants might became dead or injured. For this, the commanders used to follow a strict discipline in which they were obliged to replace the slain soldiers with fresh forces of the reserved units and expedite the wounded warriors for medical treatments, after which they used to go back to their units as soon as they were healed. However, the total number of this task force was intact, fixed at ten thousands soldiers, and were ready for any operation in any place and at any time.

Practically speaking the Persec code resembles closely to the procedure applied to the Immortal Army. Of this, the data (the army) that is going to be transmitted through communication medium, may get corrupted (injured warriors). After detecting the error(s), the receiver starts a procedure to repair the data by applying the appended extra bits (healing the wounded), otherwise it requests for retransmission of data (replacing demised soldiers).

According to this scenario some important keywords need to be defined:

Data-packet (*P*): The stream of data bits with a pre-specified length that should be transferred from transmitter to receiver correctly.

Data-length (*L*): The length of standard data-packet (not considering the check bits).

Base (*n*): Based on the Persec algorithm, before coding, a base number must be selected (greater than 2) which plays an important role during the detection/correction process, like a sieve to detach the affected bits. Note that, selecting a base 1 transforms our code to parity code, and any increase in base value would increase the correction rate, redundancy and calculation time.

Redundancy (*R*): Total number of extra bits that helps detection/correction processes.

Guard-bits (*gb*): A stream of *n*-1 ones (or zeroes) that must be added to the beginning and the end of data packet to avoid missing the data bits during coding procedure.

Segment (*S*): Based on Persec code algorithm, in each iteration of coding process, data-packet is split into consequent segments of size *n*. These segments can have values between 0 and 2^n-1.

Iteration (*i*): Persec algorithm performs its coding procedure *n* times (iterations) for each data-packet.

Hot-bits (*hb*): The bits which considers error prone in each iteration.

Retransmission-bits (*rb*): After detecting hot-bits in different iterations, three scenarios are possible:

1. Only one bit is singled out as hot-bit, exactly *n* times (for all iterations). This bit is the one which contains the error and have to be corrected.

2. The algorithm finds some suspected places, but none of them have been marked hot-bit for *n* times. This is the case of more than one bit error and we prove that, it is impossible for one bit error.

3. More than one bit is considered hot-bit for *n* times. These are the retransmission-bit candidates which are requested to be retransmitted by sender. Number of these bits is much less than L and the error bit is certainly one of them.

Distance (*d*): The distance between an error bit location and the end of its segment.

All of the forementioned parameters would be tuned according to algorithm manipulation, required accuracy, and calculation time.

III. PERSEC CODE

Persec code employs the same approach as in parity code, in which data-packet (usually a byte) uses a check bit which makes the sum of 1's odd/even [1]. Among the most well-known error detection methods, parity code is the best choice for small data-packets in transmission and communication environments, but not in telecommunications. Further more, Hamming code, Residue codes and also BCH codes are popular in both small and large data intercommunications and/or telecommunications. But they impose a considerably large time overload for large data-packets [18,19].

In turn, Persec code is an improved code for large data-packets in telecommunications. Since in telecommunications, usually we deal with large data-packets, so it works very well, but is not efficient for very small data-packets.

Depending on conditions, some methods may not detect or correct the errors. But Persec code has the ability of always detecting the errors, although sometimes is not able to locate them exactly. Notwithstanding, it reports a few suspected bits within which one of them is surely the error. In this case it requests the sender to retransmit these few bits again.

A. Structure of the Code

According to the algorithm, before coding the data in Persec code, a base number *n* should be selected. Afterward, the data-packet will be partitioned into several small boxes (segments) of *n*-bits.

For instance, if we have a data-packet of 64 bits with a given base of 3, then it can be split into 21 3-bit segments and one bit is out. To hinder this problem, *n*-1 bits of 0 (or 1) should be added at the beginning and also the end of the data-packet as guard-bits. Later these extra bits will be removed, after when the correction process in destination node is done. Hence, we have a data-packet of 68 bits, split into 22 segments and two redundant bits at both ends of packet.

B. Coding in Persec Code

Every *n*-bit segment of data-packet make a number in the range of 0 to 2^n-1 (by $n = 3$, an octal number between 0 and 7) and the total frequency of each number is odd or even. Hence a parity bit for each of these 2^n numbers (for $n = 3$, 8 numbers) can be assigned. For example, the following 64-bit data-packet is given, with a base of $n = 3$ and employing even parity checking we have:

111011000010101100001101100111110101110001101011
0011001110001010

Now, to align the given number, two (*n*-1) zeros is required to be attached to both ends (sides) of data-packet as guard-bits:

001110110000101011000011011001111101011100011010
11001100111000101000

Figure 1 illustrates the resultant 68-bit entity which is split into 3-bit segments and every segment represents an octal number.

Bit Offset	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Binary	0	0	1	1	1	0	1	1	0	0	0	0	1	0	1	0	1	1
Octal		1			6			6			0			5			3	

…

18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44
0	0	0	0	1	1	0	1	1	0	0	1	1	1	1	1	0	1	0	1	1	1	0	0	0	1	1
	0			3			3			1			7			5			3			4			3	

…

45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67
0	1	0	1	1	0	0	1	1	0	0	1	1	1	0	0	0	1	0	1	0	0	0
	2			6			3			1			6			1			2		-	-

Figure 1. Partitioning the data-packet and the corresponding octal numbers, (1st iteration).

At this point, the quantity of every octal number, corresponding to these 22 3-bit segments (each in the range of 0 to 7) is calculated. Table I represents the result of this calculation for 1st iteration, considering even parity checking. Here, we calculate the total of each individual octal number in the data-packet. Each sum can be either an odd or an even number, and applying even parity, just odd numbers can get signed.

TABLE I. FREQUENCY OF OCTAL NUMBERS AND THEIR PARITY BITS (TRANSMITTER SIDE).

Octal Number	Iteration 1		Iteration 2		Iteration 3	
	Sum	*Parity*	*Sum*	*Parity*	*Sum*	*Parity*
0	2	0	2	0	3	1
1	4	0	1	1	3	1
2	2	0	3	1	1	1
3	6	0	3	1	2	0
4	1	1	4	0	3	1
5	2	0	2	0	5	1
6	4	0	5	1	2	0
7	1	1	2	0	3	1

But it is not finished yet. This process must be repeated *n* times, shifting the segments to the right, one bit in each iteration. Figure 2 and Figure 3 show the results of 2nd and 3rd iterations of the example and also the parity bits besides their octal values have been shown in Table I (Iteration 2 and 3).

Obviously, at the end of coding process with base *n*, we have *n* sets of 2^n bits and therefore $n\times2^n$ parity bits.

Bit Offset	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Binary	0	0	1	1	1	0	1	1	0	0	0	0	1	0	1	0	1	1	0
Octal	-		3			5			4			1			2			6	

…

19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45
0	0	0	1	1	0	1	1	0	0	1	1	1	1	1	0	1	0	1	1	1	0	0	0	1	1	0
	0			6			6			3			7			2			7			0			6	

…

46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67
1	0	1	1	0	0	1	1	0	0	1	1	1	0	0	0	1	0	1	0	0	0
	5			4			6			3			4			2			4		-

Figure 2. Partitioning the data-packet and the corresponding octal numbers, (2nd iteration).

Bit Offset	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Binary	0	0	1	1	1	0	1	1	0	0	0	0	1	0	1	0	1
Octal	-	-		7			3			0			2			5	

…

17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43
1	0	0	0	0	1	1	0	1	1	0	0	1	1	1	1	1	0	1	0	1	1	1	0	0	0	1
	4			1			5			4			7			6			5			6			1	

…

44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67
1	0	1	0	1	1	0	0	1	1	0	0	1	1	1	0	0	0	1	0	1	0	0	0
	5			3			1			4			7			0			5			0	

Figure 3. Partitioning the data-packet and the corresponding octal numbers, (3rd iteration).

C. *Procedure of Error Detection*

To detect the errors, transmitter sends the data-packets with $n\times2^n$ extra-bits to the destination. Receiver needs a system to reproduce the extra-bits and compare that with the transmitted extra-bits. If there is no difference, it means that no error has been occurred in the transmitted data-packet. But, if difference(s) was/were detected, then it means error has been occurred. If any error is occurred on one bit, then it will change the result of calculations at the receiver side.

For example, if an error occurs on the bit number 9, the values of the redundant bits will change, as shown in Table II.

TABLE II. FREQUENCY OF OCTAL NUMBERS AND THEIR PARITY BITS (RECEIVER SIDE).

Octal Number	Iteration 1		Iteration 2		Iteration 3	
	Sum	*Parity*	*Sum*	*Parity*	*Sum*	*Parity*
0	1	0	2	0	2	0
1	4	0	1	1	3	1
2	2	0	3	1	2	0
3	6	1	3	1	2	0
4	2	0	3	1	3	1
5	2	0	3	1	5	1
6	4	0	5	1	2	0
7	1	1	2	0	3	1

When receiver compares these (old and new) parity sets, it can find out which octal number is changed to another. Table III shows the comparison results.

This means that receiver finds out a 0 from the set of octal numbers on the first iteration has changed to 4 (or a 4 has changed to 0) then a 4 from the set of octal numbers of second iteration has changed to 5 (or a 5 has changed to 4) and also a 0 from the set of octal numbers of third iteration has changed to 2 (or a 2 has changed to 0). All of these octal numbers and segments are suspected, but error has occurred only in one of them.

D. *The Error Correction Process*

Since the main focus of this paper is to study 1-bit error case, so this section presents solution for 1-bit error detection/correction. As mentioned before, errors would change the octal number of some segments. To locate the wrong bit, an XOR gate helps to find out which one is wrong.

Bit Offset	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
Hot-Bits (1st Iteration)										*hb*	.	.							*hb*	.	.										
Hot-Bits (2nd Iteration)					.	.	*hb*	.	.	*hb*																					
Hot-Bits (3rd Iteration)									.	*hb*	.	.	*hb*	.																	
Error/Retransmitted Bits										***Err***																					

Bit Offset	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67
Hot-Bits (1st Iteration)									*hb*	.	.																										
Hot-Bits (2nd Iteration)																.	.	*hb*	.	.	*hb*							.	.	*hb*				.	.	*hb*	
Hot-Bits (3rd Iteration)																													.	*hb*	.				.	*hb*	.
Error/Retransmitted Bits																																					

Figure 4. Error Resolution Table with hot-bits and the error-bit.

TABLE III. COMPARISON OF THE RECEIVED AND REPRODUCED PARITY BITS

Octal Number	Iteration 1			Iteration 2			Iteration 3		
	Tx	***Rx***	***Diff***	***Tx***	***Rx***	***Diff***	***Tx***	***Rx***	***Diff***
0	0	0	-	0	0	-	1	0	X
1	0	0	-	1	1	-	1	1	-
2	0	0	-	1	1	-	1	0	X
3	0	1	X	1	1	-	0	0	-
4	1	0	X	0	1	X	1	1	-
5	0	0	-	0	1	X	1	1	-
6	0	0	-	1	1	-	0	0	-
7	1	1	-	0	0	-	1	1	-

TABLE IV. ALL POSSIBLE CASES AND THE EXACT ERROR PLACE IN EACH CASE, FOR BASE 3.

Numbers		**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
		000	***001***	***010***	***011***	***100***	***101***	***110***	***111***
0	***000***	-	001	010	-	100	-	-	-
1	***001***	001	-	-	010	-	100	-	-
2	***010***	010	-	-	001	-	-	100	-
3	***011***	-	010	001	-	010	-	-	100
4	***100***	100	-	-	-	-	001	010	-
5	***101***	-	100	-	-	001	-	-	010
6	***110***	-	-	100	-	010	-	-	001
7	***111***	-	-	-	100	-	010	001	-

It should be noted that each octal number is represented by 3 bits, and for altered octal numbers only one bit gets corrupted (for 1-bit error case). For instance, octal number 7(111) would change to one of these 6(110), 5(101), 3(011) octal numbers but it never changes to 0(000) which is different in three bits. Therefore, when an octal number changes to another number, the difference between both numbers is exactly one bit. By bit-wise XORing of both octal numbers (original and corrupted numbers), we can find the incorrect bit number. For example, octal number 5(101) is correct and the transmitted corrupt information is 4(100), by XORing both numbers, XOR(101, 100) = 001, everybody can figure that the bit number one is wrong. It means that the zeros in XOR's output represent the correct bits and ones indicate the incorrect bit. Table IV presents all possible cases, using base 3 and exact location of error-bits.

Since every data-packet is comprised of several segments, and one octal number is assigned to each segment, so one data-packet may have more than one segment with the same octal value.

In this case, for each iteration we have more than one suspected bit which are marked as hot-bit. To locate the corrupted bit precisely, we need to provide a table which is called "Error Resolution Table" in which all of the *hb*s for each iteration are reported. The error bit get marked in all iterations (maximum) which is crucial in error location process. Figure 4 depicts the Error Resolution Table of the previous example and the correction process.

During the correction process, sometimes we encounter the situation in which after finishing all iterations in error resolution process, we end-up with more than one suspected bits in the table, but the procedure is not able of locating the error bit.

Bit Offset	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
Hot-Bits (1st Iteration)	.	*hb*	.													.	*hb*	.				.	*hb*	.	.	*hb*	.	.	*hb*	.	
Hot-Bits (2nd Iteration)														*hb*	.	.	*hb*	.	.				*hb*	.	.	*hb*	.	.			
Hot-Bits (3rd Iteration)															.	.	*hb*	.	.	*hb*				.	.	*hb*	.	.	*hb*		
Error/Retransmitted Bits																	***rb***									***rb***					

Bit Offset	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67
Hot-Bits (1st Iteration)						.	*hb*	.				.	*hb*	.							.	*hb*	.	.	*hb*	.				.	*hb*	.	.	*hb*	.		
Hot-Bits (2nd Iteration)				*hb*	.	.							*hb*	.	.							*hb*	.	.							*hb*	.	.				
Hot-Bits (3rd Iteration)					.	.	*hb*							.	.	*hb*							.	.	*hb*							.	.	*hb*			
Error/Retransmitted Bits																																					

Figure 5. Error Resolution Table, the hot-bits, and the suspected bits (retransmission-bits).

To cope with this problem, the correction procedure performs a special process in which the receiver requests the sender to retransmit only the suspected bits. This approach saves time in compare with traditional techniques which require the retransmission of whole data-packet, and this is the key feature of our method. As an example, let assume in the previous example error affects the bit 25. Figure 5, demonstrates the error resolution table. After executing the procedure, we found that there are two suspected bits, bit number 16 and 25, which must be retransmitted. Since our method is designed for large data transmission, for a 2kb data packet and tentative 4 suspected bits, only the 4 bits need to be retransmitted which demonstrates the power of this robust time-saving technique.

Later it will be illustrated that any increase in value of base (*n*) decreases the situations of retransmission-bits. Therefore, a larger base value increases the successful correction rate of Persec code.

IV. Analysis

A. Redundant Bits

A coded data-packet contains a block of *k*-bit information followed by a group of *r* check-bits. This would result in a block of $L = k + r$ bits which is called a code-word, and is referred as $[L, k]$ code [20]. As was shown in section 3, Persec code with base *n* in a similar way is a $[L, k]$ code, which:

$$L = k + (n \times 2^n) \qquad (1)$$

Here *r* or its equivalent $(n \times 2^n)$ is independent from *k*.

B. Case of Retransmission Bits

As was mentioned before, here we study the one bit error case. Of this, one can mathematically illustrate the statistical model and discover its probability function. Consequently, Persec parameters have been tuned with theoretically calculated values which best fit the real world applications (such as *L* and *n*).

Suppose that the i^{th} bit of data-packet has been affected in one iteration and some hot-bit(s) like the j^{th} bit has (have) also been marked (Figure 6).

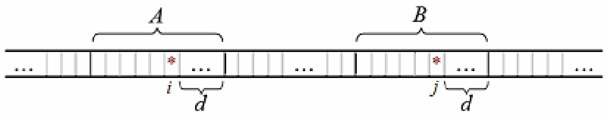

Figure 6. Error bit (i^{th}) and the hot-bit (j^{th}) position.

Note that *A* and *B* are binary values of the segments and *d* is the distance of bits *i* and *j* from the right end of segments. *A* and *B* may differ in at most one bit, which can be detected conveniently, using XOR gates. That's why the two segments have the same offset (shown by *d* in the figure), therefore the j^{th} and i^{th} bits are far from each other as much as an integer coefficient of n. Consequently, all of the segments with values *A* or *B* are suspected at the position of their d^{th} bit --from the right end of the segment-- and all of these bits are marked as hot-bit. Recall that, we have only one error, so, by this method, only one *A* converts to a *B*. As a result, receiver computes frequency of *A* one unit less than of sender's during construction of header. And for the *B*, receiver computes its frequency one unit more. So at the receiver's header, in this iteration, the bits representing the parity of *A* and *B* are complemented. This is the way receiver detects the error.

This process is applied to all iterations and eventually the bits which have been tagged as hot-bit for *n* times (in all of the iterations) are retransmission-bit candidates. By the way, these are the bits which potentially may be corrupted and one of them surly the error bit.

For further analysis, suppose that the j^{th} bit is one of the retransmission-bits but not the error bit. As depicted in Figure 7, *n*-1 bits on left and right side of the j^{th} bit must follow each other the same way as bits around the error bit (i^{th} bit).

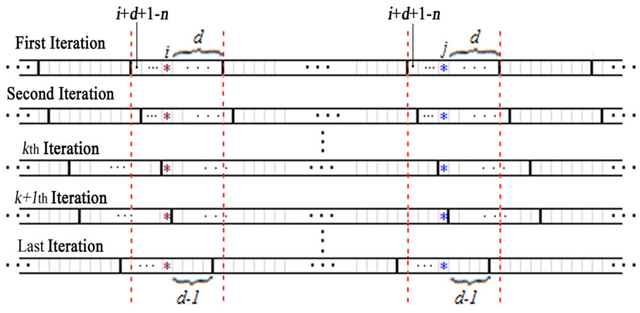

Figure 7. An error bit (i^{th}) and a retransmission-bit (j^{th}) would change in all of iterations.

Note that *k*+1 is the iteration in which the i^{th} bit changes its corresponding segment to the one next to its left side (same as j^{th} bit). We can write the following equations according to the iterations shown before:

1st iteration:

$$\{C_{i+d+1-n}, C_{i+d+2-n}, \ldots, C_{i+d}\} = \{C_{j+d+1-n}, C_{j+d+2-n}, \ldots, C_{j+d}\}$$

2nd iteration:

$$\{C_{i+d+2-n}, C_{i+d+3-n}, \ldots, C_{i+d+1}\} = \{C_{j+d+2-n}, C_{j+d+3-n}, \ldots, C_{j+d+1}\}$$

… k^{th} iteration:

$$\{C_i, C_{i+1}, \ldots, C_{i+n-1}\} = \{C_j, C_{j+1}, \ldots, C_{j+n-1}\}$$

k+1st iteration:

$$\{C_{i-n+1}, C_{i-n+2}, \ldots, C_i\} = \{C_{j-n+1}, C_{j-n+2}, \ldots, C_j\}$$

… Last iteration:

$$\{C_{i+d-n}, C_{i+d-n+1}, \ldots, C_{i+d-1}\} = \{C_{j+d-n}, C_{j+d-n+1}, \ldots, C_{j+d-1}\}$$

These equalities implies:

$$\{C_{i-n+1}, C_{i-n+2}, \ldots, C_{i+n-1}\} = \{C_{j-n+1}, C_{j-n+2}, \ldots, C_{j+n-1}\} \qquad (2)$$

Except (probably) for the C_i and C_j.

The result has been shown in Figure 8 which says that $A^l = B^l$ and $A^r = B^r$:

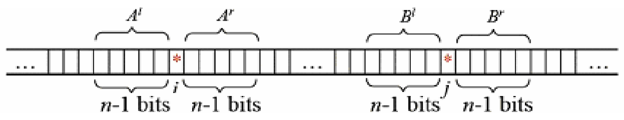

Figure 8. Retransmission-bits constraints regarding to error position.

C. *The Probability of Single Error Correction*

According to previous section, n-1 bits before and after the retransmission-bits must be one by one equal to that of the exact error bit i.e. $A^l=B^l$ and $A^r=B^r$ when A^l and A^r are binary stream values of the n-1 bits before and after the ith bit, and in the same way for the B^l and B^r (Figure 8) such that:

$$j = i \pm m.n;\ m \text{ in } \{1,2,...\}$$

If the jth bit is a retransmission-bit, then $A^l=B^l$ and $A^r=B^r$. The probability of facing such a case is:

$$\Pr(A^l = B^l) = \Pr(A^r = B^r) : \frac{1}{2^{n-1}}$$

Therefore, B is not a retransmission-bit by the following probability:

$$\Pr(A^l \neq B^l) + \Pr(A^r \neq B^r) : 1 - \frac{1}{(2^{n-1})^2} \quad (3)$$

Now, the probability function of not occurring such situation, for a L size data-packet using Persec with base n can be formulated:

$$\Pr(r = \phi) : \left(1 - \frac{1}{(2^{n-1})^2}\right)^{(\frac{L}{n}-2)} \quad (4)$$

This is the probability of correcting one bit error by the Persec code. Simulation results of Persec code strongly supports this formula, although the function has been a little overestimated (look at Table V to VIII).

TABLE V. BASE 4 RESULTS.

***L* (bit)**	***R* (%)**	**C+ Rate (Formula)**	**C+ Rate (Simulation)**	***rb* (%)**	**Time (ms)**
128	50	62.35	61.35	1.76	0.3
256	25	37.67	37.34	1.00	0.3
512	12.5	13.77	13.75	0.64	0.6
1 k	6.4	2.01	1.86	0.50	0.8
8 k	0.8	0.00	0.00	0.40	2.6
64 k	0.1	0.00	0.00	0.39	15.6

TABLE VI. BASE 6 RESULTS.

***L* (bit)**	***R* (%)**	**C+ Rate (Formula)**	**C+ Rate (Simulation)**	***rb* (%)**	**Time (ms)**
128	300	98.16	98.32	1.57	0.4
256	150	96.07	96.05	0.79	0.6
512	75	92.21	92.14	0.40	1.0
1 k	38.4	85.11	85.28	0.21	1.7
8 k	4.8	27.24	28.02	0.03	5.1
64 k	0.6	0.00	0.00	0.02	19.9

TABLE VII. BASE 8 RESULTS.

***L* (bit)**	***R* (%)**	**C+ Rate (Formula)**	**C+ Rate (Simulation)**	***rb* (%)**	**Time (ms)**
1 k	204.8	99.25	99.31	0.20	2.7
2 k	102.4	98.50	98.39	0.10	4.2
4 k	51.2	97.01	96.93	0.05	7.2
8 k	25.6	94.09	93.65	0.03	11.5
16 k	12.8	88.52	88.37	0.01	16.9
32 k	6.4	78.35	78.31	0.01	24.1
64 k	3.2	61.37	60.89	0.00	35.0

TABLE VIII. BASE 10 RESULTS.

***L* (bit)**	***R* %**	**C+ Rate (Formula)**	**C+ Rate (Simulation)**	***rb* Rate**	**Time (ms)**
1 k	1024	99.96	99.98	0.19	5.4
2 k	512	99.92	99.97	0.10	7.3
4 k	256	99.85	99.86	0.05	10.9
8 k	128	99.70	99.68	0.02	18.0
16 k	64	99.39	99.41	0.01	30.5
32 k	32	98.79	99.62	0.01	50.8
64 k	16	97.59	97.56	0.00	77.9

D. *Simulation and Statistic Results*

Every ED/CC algorithm may fall into one of the following scenarios:

1) *Always detects and corrects errors, correctly.*
2) *Detects and correct errors, incorrectly (mistakes).*
3) *Always detects, but can not correct errors (D+ C-).*

As it can be seen, according to theoretical analysis and also the simulation results, considering only one bit error, the Persec code performs great with a successful hit-rate of 100 percent of precision. Furthermore, the results show that the correction rate conforms with the outcomes of the proposed formulas. Simulations have been run on an Intel Pentium IV (Centrino - 1.7GHz) with 512MB RAM.

Figure 9 shows correction rate of the Persec code for the bases ranging from 4 to 10 and denotes that by increasing the base and decreasing the data-length, the precision of the method is increased.

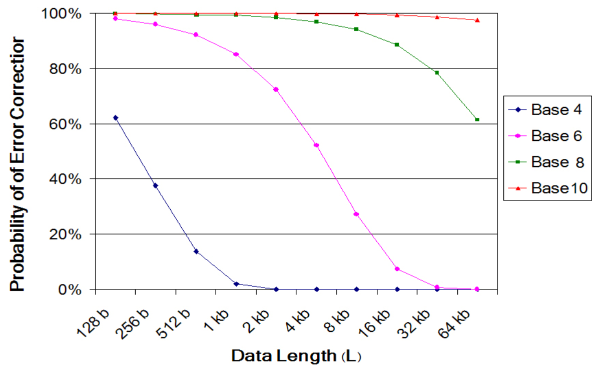

Figure 9. The probabilities of error correction.

V. Conclusion

In this paper an innovative method has been proposed for detecting and correcting errors which is inspired from parity codes, but can correct the error. The most important characteristic of this method is detection of hot-bits and recommending a set of retransmission-bits according to the value of base. This process is done when the method fails to correct the error and specifies a few bits to be retransmitted instead.

Persec code has the following advantages:

- It is an independent code, there is no relation between the length of packets and check-bits.
- It can detect burst errors.
- It can find more than one error.
- Its hardware implementation is easy.
- It is flexible to versatile conditions and different precisions.
- For the long data it requires small amount of redundancy.
- It offers real-time detection and fast correction. Figure 10 illustrates process times of Persec code for bases from 4 to 10 and the covered space for various data-lengths in contrast with Hamming and BCH codes [Hamming and BCH procedures has been reported from Reff. 21].

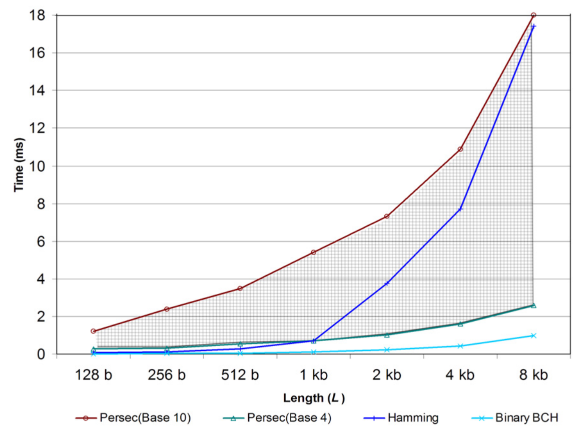

Figure 10. Process time of Persec code, Hamming code and BCH codes.

Its disadvantages are:

- On short data-packets, it imposes large time overhead.
- There is no guarantee for full correction.
- Guard-bits need protection against faults.

VI. Future Works

The following topics can be studied for further research in the subject:

- Research on multi-error correction
- Mathematical computation of hamming distance
- Finding the maximum fault coverage
- Finding the best values for parameters which best fit the situations.

References

[1] John Barry W., *Design and Analysis of Fault-Tolerant Digital Systems*, Addison-Wesley Publications, 1989.

[2] Lin Shu, *An Introduction to Error-Correcting Codes*, Prentice-Hall, 1970.

[3] Pless Vera, *Introduction to the Theory of Error-Correcting Codes*, Prentice-Hall, 1990.

[4] Hui Li, Jan Lindskog, Goran Malmgren, Gyorgy Miklos, Fredrik Nilson and Gunnar Rydnell, "Automatic repeat request (ARQ) mechanism in HIPERLAN/2," in Vehicular Technology Conference Proceedings, vol. 3 of VTC, (Tokyo), 2000, pp. 2093-2097.

[5] W. W. Peterson and D. T. Brown, "Cyclic codes for error detection," in Proceeding IRE, vol. 49, 1961, pp. 228-235.

[6] R. W. Hamming, "Error detecting and error correcting codes," Bell System Tech. J. 29, 1950, pp. 147-160.

[7] E. F. Assmus,H. F. Mattson, "On weights in quadratic-residue code"s, Discrete Mathematics, Volume 3, Issues 1-3, 1972, pp. 1-20.

[8] A.W. Nordstorm and J. P. Robinson, "An optimum nonlinear code," Information control, 1967, pp. 284-287.

[9] Claude Berrou and Alain Glavieux, "Near optimum error correcting coding and decoding: Turbo-codes," IEEE Transactions on Communication, 1996, Vol. 44, No. 10, pp. 1261–1271.

[10] Erl-Huei Lu, Yi-Chang Chen and Hsiao-Peng Wuu, "A complete decoding algorithm for double-error-correcting primitive binary BCH codes of odd m," Information Processing Letters, Volume 51, Issue 3, 1994, pp. 117-120.

[11] T. A. Gulliver, W. Lin and F. Dehne, "Fast parallel decoding of double-error-correcting binary BCH codes," Applied Mathematics Letters, Volume 11, Issue 6, 1998, pp. 11-14.

[12] Erl-Huei Lu, Shao-Wei Wub and Yi-Chang Chengb, "A decoding algorithm for triple-error-correcting binary BCH codes", Information Processing Letters, Volume 80, Issue 6, 2001, pp. 299-303.

[13] F. Barsi and P. Maestrini, "A class of multiple-error-correcting arithmetic residue codes," Information and Control, Volume 36, Issue 1, 1978, pp. 28-41.

[14] R.K. Arora and Saroj Sharma, "Correction of multiple errors and detection of additive overflow in residue code," Information and Control, Volume 39, Issue 1, 1978, pp. 46-54.

[15] Ronald Klein, Murali Varanasi and Larry Dunning, "Multiple error detection/correction using the Nordstorm-Robinson code," Proceedings of 43rd IEEE Midwest Symposium on Circuits and Systems, 2000, pp. 254-257.

[16] Boris Polianskikh and Zeljko Zilic, "Induces error-correcting code for 2-bit-per-cell multi-level Dram," IEEE transaction on coding, 2000, pp. 352-355 .

[17] Tanenbaum Andrew S., *Computer Networks*, 4th Edition, Prentice-Hall, 2003.

[18] Shu Lin and E.J. Weldon, "Long BCH codes are bad," Information and Control Volume 11, Issue 4, 1967, pp. 445-451.

[19] G. Solomon, "Golay encoding/decoding via BCH-Hamming," Computers & Mathematics with Applications, Volume 39, Issue 11, 2000, pp. 103-108.

[20] Anand Srivastava, Subrat Kar and V.K. Jain, "Forward error correcting codes in fiber-optic synchronouscode-division multiple access networks," Optics Communications, Vol. 202, 2002, pp. 287–296.

[21] R. H. Morelos Zaragoza, *The Art of Error Correcting Coding*, 2nd Edition, John Wiley & Sons, 2006.