Effective method for coding and decoding RS codes using SIMD instructions

Aleksei Marov, *Researcher,* R&D department
Raidix LLC, and
PhD Student, St.Petersburg State University
Saint Petersburg, Russia
marov.a@raidixstorage.com

Sergei Platonov, *Head,* R&D department
Raidix LLC
Saint Petersburg, Russia
platonov.s@raidixstorage.com

***Abstract*—a method is introduced for efficient encoding and decoding of the Reed-Solomon codes based on the matrix formalism. In this ideology, a method is suggested for vectorization of the Berlekamp-Massey algorithm for detecting and correcting several silent data corruptions. The results of comparison of suggested method with other knows ways of decoding RS codes are presented. This approach requires small amount of additional memory for intermediate computations, can be implemented with the high speed libraries performed Galois field arithmetic, and shows promising prospects for parallelization.**

Keywords—Reed-Solomon codes, silent data corruption, Berlekamp-Massey algorithm, SIMD instructions.

I. INTRODUCTION

For several years there has been explosive growth in the amount of stored data. Software for Scale-Out Storage Systems such as HDFS [1], GFS [2], Ceph [3], allows one to create storage capacity of several tens of petabytes. Large data centers have hundreds and even thousands of data nodes using commodity hardware. Fault tolerance of such systems is performed by repeated replication and erasure coding.

Erasure coding has a much lower storage overhead compared to data replication, but has several drawbacks: the high cost of data recovery and poor performance of coding and decoding as well as a large amount of data transferred between nodes in the encoding [4]. The approaches similar to Local Reconstruction Codes [5] and Pyramidal Codes [6] are used to resolve issues of recovery cost.

The performance issues of erasure coding, and, in particular, operations in Galois fields, can be tackled by utilization of the 128-bit instructions of modern CPU, such as Intel's Streaming SIMD Extensions [7, 19].

Erasure codding solutions are exploited in Microsoft Azure [5], Google GFS [8], HDFS-RAID, etc. However, these solutions protect only against complete disk failures and latent sector errors. One of the most vital problems appeared in the practice of the massive data storages is Silent Data Corruption [9]. Several manufacturers of Data Storage Systems offer solutions to protect against Silent Data Corruptions, based on the complement of ECC to data blocks and the local checksum computations [10].

In the present paper we discuss algorithms (and some aspects of their implementation) for distributed error-correcting coding and archival storage systems using Reed-Solomon codes. They are focused on detecting and correcting several silent data corruptions.

II. ARITHMETIC OF THE GALOIS FIELDS

The arithmetic of Galois fields is often used in the error-correcting coding. This is done in order not to deal with individual bits of information during encoding, but to work with entire codewords. We consider here some basic concepts of the Galois field theory.

Galois field of characteristic 2 is the field of 2^q elements $GF(2^q) = \mathbb{Z}_2[x]/\langle f(x)\rangle$ where $f(x)$ stands for an irreducible over $\mathbb{Z}_2$ polynomial of the degree q. An element of this field can be treated either as a q-bit codeword or as a polynomial in the variable x of degree less than q with the coefficient vector coinciding with the codeword. For instance, the codeword (10000101) and the polynomial $x^7 + x^2 + 1$ are the equivalent representations of a single element of $GF(2^8)$.

For the elements $a(x)$ and $b(x)$ of the field, the addition operation is defined as the sum $a(x) + b(x)$ with reduction coefficients modulo 2. Multiplication operation $a(x) * b(x)$ is defined as the product of polynomials modulo $f(x)$ with reduction coefficients modulo 2. Inverse element for the nonzero element $a(x) \in GF(2^q)$ is an element $a^{-1}(x)$ of the field such that $a(x) * a^{-1}(x) = 1$; here 1 denotes the neutral element with respect to multiplication.

An essential property of the Galois field is the existence of a primitive element which hereinafter will be denoted by a. This element has the property that all its powers a^0, $a^1, \dots$, a^{2^q-2} represent distinct nonzero elements of the field.

III. CODING AND DECODING USING THE VANDERMONDE MATRIX

One of the easiest and most convenient ways to calculate m checksums is based on the usage of the Vandermonde matrix. Properties of the matrix can uniquely recover data when decoding.

A. Vandermonde Matrix and Its Inverse

Consider the general Vandermonde matrix in the form

The first author was partially supported by the St. Petersburg State University research grant #9.38.674.2013.

$$V_{m,n}(\lambda_1,\dots,\lambda_n)=\begin{pmatrix}1 & 1 & \cdots & 1\\ \lambda_1 & \lambda_2 & \cdots & \lambda_n\\ \vdots & \vdots & \cdots & \vdots\\ \lambda_1^{m-1} & \lambda_2^{m-1} & \cdots & \lambda_n^{m-1}\end{pmatrix}$$

with the entries $\lambda_1,\lambda_2,\dots,\lambda_n$ called its generating elements. It is well-known that

$$det\, V_{n,n}(\lambda_1,\lambda_2,\dots,\lambda_n)=\prod_{1\le i<j\le n}(\lambda_j-\lambda_i)$$

and, therefore, a square matrix $V_{n,n}(\lambda_1,\lambda_2,\dots,\lambda_n)$ is invertible if and only if its generating elements are distinct . The inverse matrix can be found either by using usual matrix inversion methods (such as Gauss-Jordan, LUP decomposition etc.) or, alternatively, by using the following result from the theory of polynomial interpolation [13, 14].

We introduce the notation

$$W(\mathfrak{x})=\prod_{i=1}^{n}(\mathfrak{x}-\lambda_i),\qquad W_j(\mathfrak{x})=\frac{W(\mathfrak{x})}{\mathfrak{x}-\lambda_j},\qquad j=\overline{1,n}$$

and define the basic interpolation polynomials

$$\widetilde{W}_j(\mathfrak{x})=\frac{W_j(\mathfrak{x})}{W_j(\lambda_j)}=\frac{W_j(\mathfrak{x})}{W'(\lambda_j)} \tag{1}$$
$$=\frac{(\mathfrak{x}-\lambda_1)(\mathfrak{x}-\lambda_2)\dots(\mathfrak{x}-\lambda_{j-1})(\mathfrak{x}-\lambda_{j+1})\dots(\mathfrak{x}-\lambda_n)}{(\lambda_j-\lambda_1)(\lambda_j-\lambda_2)\dots(\lambda_j-\lambda_{j-1})(\lambda_j-\lambda_{j+1})\dots(\lambda_j-\lambda_n)}=$$
$$=w_{j,0}+w_{j,1}\mathfrak{x}+\dots+w_{j,n-1}\mathfrak{x}^{n-1}$$

Theorem 1. *The inverse of a square Vandermonde matrix* $V_{n,n}(\lambda_1,\lambda_2,\dots,\lambda_n)$ *can be found in the form*

$$\left[V_{n,n}(\lambda_1,\lambda_2,\dots,\lambda_n)\right]^{-1}=\left[w_{j,k-1}\right]_{j,k=1}^{n}=\begin{pmatrix}w_{1,0} & w_{1,1} & \cdots & w_{1,n-1}\\ w_{2,0} & w_{2,1} & \cdots & w_{2,n-1}\\ \vdots & \vdots & \cdots & \vdots\\ w_{n,0} & w_{n,1} & \cdots & w_{n,n-1}\end{pmatrix} \tag{2}$$

The proof evidently follows from

$$\widetilde{W}_j(\lambda_k)=\begin{cases}1 \text{ if } j=k\\ 0 \text{ if } j\ne k\end{cases}$$

B. Coding procedure

Using Vandermonde matrix the coding may be performed as follows: [15,16]. Suppose the q -bit data blocks $D_0,D_1,\dots,D_{n-1}$ are given. In the case of coding in data storage, these blocks are chosen from different drives and compose a stripe. Consider them as elements of the field $GF(2^q)$. Then the calculation of checksums $\{S_0,S_1,\dots,S_{m-1}\}\in GF(2^q)$ is organized as follows:

$$\begin{pmatrix}S_0\\ S_1\\ S_2\\ \vdots\\ S_{m-1}\end{pmatrix}=\begin{pmatrix}1 & 1 & \cdots & 1\\ a^{n-1} & a^{n-2} & \cdots & 1\\ a^{2(n-1)} & a^{2(n-2)} & \cdots & 1\\ \vdots & \vdots & \vdots & \vdots\\ a^{(m-1)(n-1)} & a^{(m-1)(n-2)} & \cdots & 1\end{pmatrix}\begin{pmatrix}D_0\\ D_1\\ D_2\\ \vdots\\ D_{n-1}\end{pmatrix}$$

or, equivalently, in the matrix form:

$$S_{m,1}=V_{m,n}(a^{n-1},a^{n-2},\dots,1)D_{n,1}$$

After calculating the checksums, the data stripe $(D_0,D_1,\dots,D_{n-1},S_0,S_1,\dots,S_{m-1})$ is written to the storage device, and, in case of failure of some composing blocks, they can be recovered with the aid of checksums recalculation.

C. Decoding procedure

Let the drive failure has been occurred. If the number of failed drives is less than the number of checksums, the lost data can be trustworthy recovered. Suppose the number of failed blocks equals l ($l\le m$) and their positions $k_1,\dots,k_l$ are known. Denote by

$$\overline{D}_{l,1}=[D_i]_{l,1},\ i=\{k_1,\dots,k_l\}$$

the column vector of failed blocks and by

$$D'_{n-l,1}=[D_i]_{(n-l),1},\ i=\{0,\dots,n-1\}\backslash\{k_1,\dots,k_l\}$$

the column vector of not failed blocks.

Then, the data recovery process can be described as follows:

(a) Recalculate the syndromes skipping failed blocks

Given the notation, it can be written as

$$\tilde{S}_{l,1}=V_{l,n-l}(a^{n-1},a^{n-2},\dots,a^{n-k_i},a^{n-k_i-2},\dots,a,1)D'_{n-l,1}. \tag{3}$$

Here $\tilde{S}_{l,1}$ denotes the column vector for syndromes calculated for the values of not failed blocks.

Not to be confused we call checksums the data blocks stored on drives and identified by S_i, and syndromes are values which are calculated in case of procedures of data recovery and SDC detection called. Syndromes are identified by $\tilde{S}_i$, and they aren't stored on the drives. We will use these terms for further convenience. Sometimes formulas for checksums and syndromes are the same, sometimes they are different, and it's depending on coding and decoding methods.

(b) Then the failed drives can be found as the solution of the system

$$V_{l,l}(a^{n-k_1-1},a^{n-k_2-1},\dots,a^{n-k_l-1})\overline{D}_{l,1}=\tilde{S}_{l,1}+S_{l,1} \tag{4}$$

The column of the failed blocks can be recovered by the formula

$$\overline{D}_{l,1}=\left[V_{l,l}(a^{n-k_1-1},a^{n-k_2-1},\dots,a^{n-k_l-1})\right]^{-1}(\tilde{S}_{l,1}+S_{l,1})$$

For $l\le m<2^q$ the inverse matrix at the right-hand side exists.

If some the checksums blocks are denied, in the system (4) the equations corresponding to these checksum will not appear. The system is also solvable, and failed data blocks can be restored.

IV. Types of Data Corruption in Storage Systems

We distinguish two cases of data corruption. The first one relates to such a failure of a drive or of its part, which can be detected by technical (hardware) control devices. In this case, the position of data corruption is known and is usually referred to as the *refusal* or the *failure* of the drive. In the second case, the data are distorted in the process of recording, storing or reading, but the fact of the corruption as well as its position is not detected by technical (hardware) facilities. This case is usually referred to as the *silent data corruption* (SDC) or erasures [16, 17]. In this case, the task is to establish the fact of data corruption (distortion), to find its position and to recover the data on the basis of redundant information.

In case of a small number of checksums (for instance, two as in RAID-6), a single SDC can be found as described in [11]. However, it might happen that the SDC can be detected at the moment of data reconstruction, and this will cause the restoration errors. In this case, a good solution would be to use the Sector-Disk erasure codes [18]. However, this approach requires more additional memory than Reed-Solomon codes for storing checksums and protects only against one SDC

(which is called sector error in [18]). Also, if the system is subject to a lot of SDCs and failures, the effective way is to use Reed-Solomon codes (RS codes).

V. The structure of the Reed-Solomon code

General scheme of RS codes in a systematic coding can be described as follows [16, 17].

The generating polynomial of RS code is a polynomial over the field $GF(2^q)$ of the form

$$g(\mathfrak{x}) = (\mathfrak{x} + a^{l_0})(\mathfrak{x} + a^{l_0+1}) \dots (\mathfrak{x} + a^{l_0+m-1}).$$

We denote hereinafter by $\mathfrak{x}$ the polynomial variable in polynomials over $GF(2^q)$, in order to distinguish it from x as a variable in polynomial over $GF(2)$, i.e. in the polynomial representation of the element of $GF(2^q)$.

From the two known coding procedures, namely, systematic and non-systematic, we choose the first one as being more *user-friendly*.

Consider the stripe of informational codewords $D_{n,1} = (D_0, D_1, \dots, D_{n-1})$, $D_i \in GF(2^q)$. Consider the polynomial over $GF(2^q)$

$$G(\mathfrak{x}) = D_0\mathfrak{x}^{n-1} + D_1\mathfrak{x}^{n-2} + \dots + D_{n-2}\mathfrak{x} + D_{n-1}$$

generated by the stripe.

Assuming that the code should be resistant to the failure of up to l blocks and up to p SDCs, one should organize calculation (and storage) of at least $m = l + 2p$ checksums. For this aim, the remainder polynomial $C(\mathfrak{x})$ of the division of

$$G^*(\mathfrak{x}) = G(\mathfrak{x})\mathfrak{x}^m = D_0\mathfrak{x}^{n-1+m} + D_1\mathfrak{x}^{n-2+m} + \dots + D_{n-2}\mathfrak{x}^{m+1} + D_{n-1}\mathfrak{x}^m$$

by $g(\mathfrak{x})$ is computed, so one has: $G^*(\mathfrak{x}) = Q(\mathfrak{x})g(\mathfrak{x}) + C(\mathfrak{x})$. Here $\deg G^*(\mathfrak{x}) = m + n - 1$, $\deg Q(\mathfrak{x}) = n - 1$, $\deg g(\mathfrak{x}) = m$, $\deg C(\mathfrak{x}) \le m - 1$. The coefficients of $C(\mathfrak{x})$

$$C(\mathfrak{x}) = C_0\mathfrak{x}^{m-1} + C_1\mathfrak{x}^{m-2} + \dots + C_{m-1}$$

are taken as checksums $C_{m,1} = (C_0, C_1, \dots, C_{m-1})$. Final code polynomial is chosen in the form $\bar{G}(\mathfrak{x}) = G^*(\mathfrak{x}) + C(\mathfrak{x})$ and the stripe is formed from its coefficients

$$(Y_0, \dots, Y_{N-1}) = (D_0, D_1, \dots, D_{n-1}, C_0, \dots, C_{m-1}); \qquad (5)$$

here $N = n + m - 1$. For the polynomial $\bar{G}(\mathfrak{x})$ the following equalities are valid

$$\bar{G}(1) = 0, \bar{G}(a) = 0, \dots, \bar{G}(a^{m-1}) = 0. \qquad (6)$$

These conditions are equivalent to

$$\sum_{i=0}^{N-1} Y_i a^{j(N-i-1)} = 0, \ j = \overline{0, m-1} \qquad (7)$$

and mean that syndromes computed for the undamaged stripe have zero values.

Let the stripe be damaged and this fact is discovered from the condition that at least one of the equalities (6) is not fulfilled, i.e. at least one of the values

$$\bar{G}(1) = \tilde{S}_0, \bar{G}(a) = \tilde{S}_1, \dots, \bar{G}(a^{m-1}) = \tilde{S}_{m-1}$$

is nonzero.

These values can be utilized in the error correction algorithm similar to that described in Section III *C*. For the problem of the SDC correction, they are used for constructing the error locator polynomial. On evaluating the roots of this polynomial, one can determine the SDC positions, and then to restore the original values of the damaged blocks from the corresponding system of linear equations. Some extra facts on the error locator polynomial construction can be found in [16, 17]; we will also discuss some aspects of this problem in foregoing sections.

VI. Encoding via polynomial interpolation matrix

Finding the checksums within the procedure outlined in the previous section, is rather complicated. We intend to suggest a more effective coding algorithm.

Rewrite the equations (7) with the aid of Vandermonde matrix

$$V_{m,n+m}(a^{n+m-1}, a^{n+m-2}, \dots, a, 1)\begin{bmatrix} D_{n,1} \\ C_{m,1} \end{bmatrix} = \mathbb{0}_{m,1} \qquad (8)$$

Here $D_{n,1} = (D_0, \dots, D_{n-1})^T$, $C_{m,1} = (C_0, \dots, C_{m-1})^T$, and $\mathbb{0}_{m,1}$ is a zero column vector. From (8) it follows that

$$V_{m,n}(a^{n+m-1}, a^{n+m-2}, \dots, a^m)D_{n,1} + V_{m,m}(a^{m-1}, a^{m-2}, \dots, 1)C_{m,1} = \mathbb{0}_{m,1}$$

Resolve this equation with respect to $C_{m,1}$:

$$C_{m,1} = [V_{m,m}(a^{m-1}, \dots, 1)]^{-1}V_{m,n}(a^{n+m-1}, \dots, a^m)D_{n,1} \qquad (9)$$

The inverse of the Vandermonde exists if $m < 2^q$; and one can find it with the aid of Theorem 1:

$$[V_{m,m}(a^{m-1}, \dots, 1)]^{-1} = \left[w_{j,k-1}\right]_{j,k=1}^{m}$$

where $w_{j,i}$ stand for the coefficients of the j-th basic interpolation polynomial generated by $\lambda_1 = a^{m-1}, \lambda_2 = a^{m-2}, \dots, \lambda_m = 1$. Thus, the matrix from the right-hand side of (9) can be represented as

$$\left[V_{m,m}(a^{m-1}, \dots, 1)\right]^{-1}V_{m,n}(a^{n+m-1}, \dots, a^m) = \begin{pmatrix} w_{1,0} & w_{1,1} & \cdots & w_{1,m-1} \\ w_{2,0} & w_{2,1} & \cdots & w_{2,m-1} \\ \vdots & \vdots & \cdots & \vdots \\ w_{m,0} & w_{m,1} & \cdots & w_{m,m-1} \end{pmatrix}\begin{pmatrix} 1 & 1 & \cdots & 1 \\ a^{N-1} & a^{N-2} & \cdots & a^m \\ a^{2(N-1)} & a^{2(N-2)} & \cdots & a^{2m} \\ \vdots & \vdots & \vdots & \vdots \\ a^{m(N-1)} & a^{m(N-2)} & \cdots & a^{mm} \end{pmatrix}$$

Multiply the first row of the left matrix by the first column of the right matrix

$$(w_{1,0}, w_{1,1}, \dots, w_{1,m-1})(1, a^{N-1}, \dots, a^{m(N-1)})^T = w_{1,0} + w_{1,1}a^{N-1} + \dots + w_{1,m-1}a^{m(N-1)} = \widetilde{W}_1(a^{N-1})$$

i.e. the result of the multiplication equals the value of the first basic interpolation polynomial. Similar arguments are valid for other multiplication results and from the formula (9) it follows the coding algorithm

$$C_{m,1} = \begin{pmatrix} \widetilde{W}_1(a^{N-1}) & \widetilde{W}_1(a^{N-2}) & \cdots & \widetilde{W}_1(a^m) \\ \widetilde{W}_2(a^{N-1}) & \widetilde{W}_2(a^{N-2}) & \cdots & \widetilde{W}_2(a^m) \\ \vdots & \vdots & \vdots & \vdots \\ \widetilde{W}_m(a^{N-1}) & \widetilde{W}_m(a^{N-2}) & \cdots & \widetilde{W}_m(a^m) \end{pmatrix}D_{n,1} \qquad (10)$$

We called matrix in (10) as the *Coding matrix*. If this matrix used for checksums calculation than conditions (7) are satisfied. This encoding method has that advantage that the Coding matrix is fixed for any sequence of encoded blocks and therefore can be precomputed. It is also possible to speed up coding computation by using the process of multiplication in a

Galois field using the method described in [7]. In this method, the x86 assembly instruction PSHUFB can be used for multiplication of the vector containing several elements of the field by the fixed element of the field. This instruction corresponds to the intrinsic function of C programming language _mm_shuffle_si128i(…) [19], which, as one of the arguments, has the mask corresponding to the multiplier element. Since the coding matrix is fixed, it is efficient not to perform the calculation of these masks for multiplication, but to store them and to use for fast multiplication within the encoding process.

VII. Decoding via recovery matrix

One can use the properties of the Vandermonde matrix when decoding data. Assume that for data blocks $D_0, D_1, \dots, D_{n-1}$ the checksums $C_0, C_1, \dots, C_{m-1}$ are calculated by the algorithm from Section VI. Suppose that within the stripe (5) $l \le m$ damages has been occurred at the known positions $k_1, \dots, k_l$. Denote by

$$\bar{Y}_{l,1} = [Y_i]_{l,1},\ i = \{k_1, \dots, k_l\}$$

the column vector of failed blocks and by

$$Y'_{(N-l),1} = [Y_i]_{(N-l),1},\ i = \{0, \dots, N-1\} \backslash \{k_1, \dots, k_l\}$$

the column vector of not failed blocks.

Then the decoding algorithm can be described as follows:

(a) Calculate the syndromes by the formulas (7), skipping the failed blocks in the calculation process. In matrix form, this can be written as

$$\tilde{S}_{l,1} = V_{l,N-l}(a^{N-1}, a^{N-2}, \dots, a^{N-k_i}, a^{N-k_i-2}, \dots, a, 1) Y'_{(N-l),1} \quad (11)$$

(b) Restore failed blocks using inversion of the Vandermonde matrix

$$\bar{Y}_{l,1} = \left\{V_{l,l}(a^{N-k_1-1}, a^{N-k_2-1}, \dots, a^{N-k_l-1})\right\}^{-1} \tilde{S}_{l,1} \quad (12)$$

(c) Substitute (11) *into* (12):

$$\bar{Y}_{l,1} = \left\{V_{l,l}(a^{N-k_1-1}, \dots, a^{N-k_l-1})\right\}^{-1} * V_{l,N-l}(a^{N-1}, a^{N-2}, \dots, a^{N-k_i}, a^{N-k_i-2}, \dots, a, 1) Y'_{(N-l),1}$$

Denote the matrix in the right-hand side of the last formula as $R_{l,(N-l)}$. Using the arguments from Section III *A*, one gets

$$R_{l,(N-l)} = \begin{pmatrix} w_{1,0} & w_{1,1} & \cdots & w_{1,l-1} \\ w_{2,0} & w_{2,1} & \cdots & w_{2,l-1} \\ \vdots & \vdots & \cdots & \vdots \\ w_{l,0} & w_{n,1} & \cdots & w_{l,l-1} \end{pmatrix} V_{l,N-l}(a^{N-1}, a^{N-2}, \dots, a^{N-k_i}, a^{N-k_i-2}, \dots, a, 1)$$

where $w_{j,i}$ are the coefficients of the j-th basic interpolation polynomial $\widetilde{W}_j$ generated by $\lambda_1 = a^{N-k_1-1}, \dots, \lambda_l = a^{N-k_l-1}$ (v. Section III *A*). Following further the reasons from Section VI, one can represent this matrix as

$$R_{l,(N-l)} = \begin{pmatrix} \widetilde{W}_1(a^{N-1}) & \widetilde{W}_1(a^{N-2}) & \cdots & \widetilde{W}_1(a^{N-k_i}) & \widetilde{W}_1(a^{N-k_i-2}) & \cdots & \widetilde{W}_1(1) \\ \widetilde{W}_2(a^{N-1}) & \widetilde{W}_2(a^{N-2}) & \cdots & \widetilde{W}_2(a^{N-k_i}) & \widetilde{W}_2(a^{N-k_i-2}) & \cdots & \widetilde{W}_2(1) \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \widetilde{W}_l(a^{N-1}) & \widetilde{W}_l(a^{N-2}) & \cdots & \widetilde{W}_l(a^{N-k_i}) & \widetilde{W}_l(a^{N-k_i-2}) & \cdots & \widetilde{W}_l(1) \end{pmatrix}$$

or, in compact form,

$$R_{l,(N-l)} = \left[\widetilde{W}_i(a^{N-j-1})\right], \qquad i = \overline{1,l}, \qquad j = \{0, \dots, N-1\} \backslash \{k_1, \dots, k_l\}$$

The corrupted blocks can be recovered as follows

$$\bar{Y}_{l,1} = R_{l,(N-l)} Y'_{(N-l),1} \quad (13)$$

We will call the introduced matrix $R_{l,(N-l)}$ as the *Recovery matrix*. The suggested method of recovery possesses an advantage that the recovery of the corrupted blocks is executed directly from those not corrupted and does not require an extra memory for storage of intermediate calculations. It should also be noted that when restoring data, (for instance, in storage) calculation of the recovery matrix $R_{l,(N-l)}$ is performed once and for all the stripes to be restored. In this regard, after the calculation of the matrix entries the masks can be used for fast multiplication as described in [7]. This will speed up the recovery process.

Described coding and decoding methods correctly restore data if $m + n < 2^q$, i.e. if the total number of blocks does not exceed the order of the field. Therefore, for larger values of m and n one needs to deal with the higher order fields.

VIII. Detection of SDC

As has been already mentioned above, the usage of the RS codes is justified by the urgent need for the SDC detection. We will present the algorithm aimed at this goal. Assume that for the data blocks $D_0, D_1, \dots, D_{n-1}$ the checksums $C_0, C_1, \dots, C_{m-1}$ were calculated, for example, with the aid of the method outlined in Section VI. Suppose that the stripe (5) contains $p \le \lfloor m/2 \rfloor$ SDCs placed at the (a priory unknown) positions $j_1, j_2, \dots, j_p$. Even the fact of the SDC presence is not known. The next we will describe two cases. In the first case there are only silent corruptions and there are not failures, and in the second case there are both types of errors.

A. Detection of SDC in the absence of failures

At first, we consider the simpler case when all the possible data damages are of the type SDC, not failures. Then we start with the problem of the SDC detection. The checksums C_i have been calculated in such a way that, in the absence of errors, the conditions (6) are fulfilled. Therefore, for detecting the presence of the SDC, one may recalculate the values for syndromes $\tilde{S}_j$

$$\tilde{S}_j = \sum_{i=0}^{N-1} Y_i a^{j(N-i-1)},\ j = \overline{0, m-1} \quad (14)$$

If at least one syndrome $\tilde{S}_j$ is not 0, then the considered stripe is corrupted. Next, we need to find the SDC positions. For this aim, construct the so-called error locator polynomial, i.e. the polynomial $\sigma(\mathfrak{x}) = \sigma_0 + \sigma_1 \mathfrak{x} + \sigma_2 \mathfrak{x}^2 + \dots + \sigma_p \mathfrak{x}^p$ with the roots coinciding with the elements $a^{N-j_1-1}, \dots, a^{N-j_p-1}$ (so, the exponents of the powers give one the SDC locations). The coefficients of this polynomial are obtained from the syndrome values $\tilde{S}_j$ by the iterative Berlekamp-Massey algorithm (BM Algorithm) [17,20,21]. The iterations are organized by the degree of this polynomial, or, in the other words, in the number of potential SDCs.

Sequence $\tilde{S}_{m-1}, \tilde{S}_{m-2}, \dots, \tilde{S}_1, \tilde{S}_0$ is an input of the algorithm, while its output is the sequence $\sigma_0, \sigma_1, \dots, \sigma_p$ of coefficients of the error locator polynomial. On evaluating it, one can find its roots $a^{N-j_1-1}, \dots, a^{N-j_p-1}$, with the aid of, for instance, the procedure by Chien [17]. The exponents of the primitive element give one the SDC locations.

When all these SDC locations are identified, the data in corrupted blocks are restored using, for instance, the algorithm from Section VII.

B. *Detection of SDC in the presence of failures*

Consider now the general case, when, in addition to the SDC, stripe contains also failures, i.e. damages with a priori known locations. This case is typical for the system at the moment of reconstruction, when necessary to ensure the correctness of the recovery, as if silent corruptions were not detected in a timely manner, at the time of reconstruction can lead to data loss. Suppose that within the stripe (5) we have l failures at the known positions $k_1, \dots, k_l$ and p SDCs at the a priori unknown positions $j_1, j_2, \dots, j_p$ (even the fact of the SDC presence is unknown). In this case, if the condition $l + 2p \le m$ is valid, then all the SDCs can be detected and all the lost data can be corrected. For this aim one should:

(a) Calculate syndromes

$$\tilde{S}_{m-1,1} = V_{m,N-l}(a^{N-1}, a^{N-2}, \dots, a^{N-k_i}, a^{N-k_i-2}, \dots, a, 1)Y'_{(N-l),1}$$

(b) Construct auxiliary polynomial

$$z(\mathfrak{x}) = \prod_{i=1}^{l}(\mathfrak{x} + a^{N-k_i-1}) = z_0 + z_1\mathfrak{x} + \cdots + z_l\mathfrak{x}^l$$

(c) Calculate values

$$T_i = \sum_{j=0}^{l} z_j \tilde{S}_{i+j}, \qquad i = \overline{0, m-l-1}$$

(d) If $T_i \neq 0$ *for at least one i then SDC has been occurred.*

(e) Error locator polynomial is constructed and the positions of SDCs are determined via the BM algorithm, in which the input sequence is chosen as $T_{m-l-1}, T_{m-l-2}, \dots, T_0$.

(f) All data are restored according to decoding algorithm from Section VII.

To detect SDC in the case of failed drives, performed T_i calculation in order to eliminate the influence of failed drives in the construction of the error locator polynomial.

IX. Using SIMD instructions in the BMA algorithm

As one may notice, the cornerstone for solving the SDC detection problem is the BM algorithm. This algorithm is rather complicated in realization, since it requires operations with polynomials over $GF(2^q)$. However, it is possible to speed up this algorithm using sse / avx SIMD extension of processor. Indeed, the basic steps of the BMA algorithm are the multiplications of a polynomial over $GF(2^q)$ by an element of the field and by the variable $\mathfrak{x}$. To implement these basic functions, it is suggested to store all the coefficients these polynomials and error locator polynomial σ on sse / avx registers. For example, if the field is chosen to be $GF(2^8)$ then one sse register can contain 16 elements of the field, i.e. it can store the polynomial of up to the 15th degree. If one wants to store polynomials of higher degrees, then it is necessary to use more than one register to store its coefficients. Within this approach, the coefficients of multiplication of a polynomial by a constant can be performed using fast multiplication [7] based on the usage of instructions _mm_shuffle_epi8(...). A multiplication of a polynomial by the variable $\mathfrak{x}$ corresponds to a shift register left by 1 byte; for this operation intrinsic function _mm_bslli_si128(…) can be used. However, the difficulties arise when the degree of polynomials becomes greater than the size of sse register, and, therefore, to store it, more than one register is needed. In this case, the multiplication by the variable requires the transfer of the high-order byte of one of the registers into the low-order byte of another register and left shifting of registers with coefficients. The transfer of a byte from one register to another can be effectively organized with the use of intrinsic functions _mm_insert_epi8(...) and _mm_extract_epi8 (...). This is illustrated in Fig. 1.

This approach allows one to significantly speed up the BM algorithm, compared with implementations that do not use vectorization. The number of CPU cycles was measured, required for constructing the error locator polynomials of different degrees. For each value of degree about 10,000 of tests were produced; the figures reflect the averaged values. It turns out that using vector version of the BMA algorithm is 2-3 times faster than in non-vector implementation (v. Fig. 2). Using the vector instructions on the stage of the error locator polynomial construction, increases significantly the speed of the SDC detection during decoding.

X. Comparison of recovery algorithms

Next we performed comparison of different decoding algorithms of Reed-Solomon codes.

1. Decoding using polynomial interpolation method described in Section VII. It should be noted that the construction of the recovery matrix *R* when decoding can also be performed using SIMD instructions. One of the advantages of this method is that the failed data are recovered from the not failed, without storing any intermediate values. In Fig. 3.1 and 3.2 the corresponding results are designated as Recovery_R.

2. Decoding by the sequential application of the formulas (11) and (12). The method consists of the two steps: the syndrome calculation, and then their values are multiplied by the Vandermonde inverse matrix. This method requires an additional memory for the intermediate storage syndromes $\tilde{S}_j$. But on the both steps of the algorithm the Fast Fourier Transformation can be applied [22]. One the other hand, one can optimize syndrome computation and further multiplication on the inverse matrix. In Fig. 3.1 and 3.2 are designated as Recovery_S.

Fig. 1. Transferring bytes from one register to another, when multiplied by $\mathfrak{x}$ for the case when polynomial degree exceeds the size of register

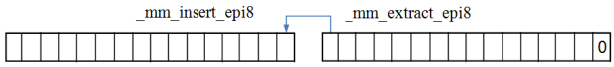
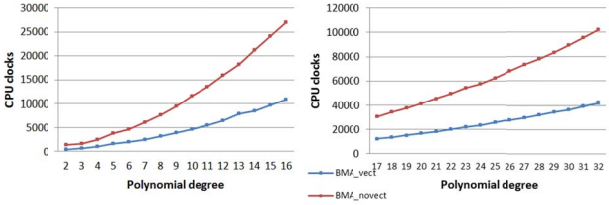


Fig. 2. Performance comparison of the no vector (blue) and vector (red) implementation of BMA algorithm



3. Using the Forney algorithm [16, 17, 23]. In Fig. 3.1 and 3.2 are designated as Recovery_F.

To measure the speed of decoding algorithms, the 4096-byte width stripe was reserved in RAM with the number of blocks corresponding to different combinations of n and m.The maximum possible amount of the failed blocks was selected and the data on them were deleted, then decoding was performed. This test was executed 10000 times on a single computing core, processor Intel Core i7-2600 CPU 3.40GHz and speed values obtained were averaged for all measurements. The operations of addition and multiplication were implemented in the field $GF(2^8)$ with the multiplication techniques described in [7]. Encoder and decoder have been written in the C programming language, as a compiler used gcc 4.7., with optimization level -O2.

Figures 3.1 and 3.2 demonstrate the measurement results. In Fig. 3.1, the value for n is assigned to $n = 128$ and the values for m is varied from 2 to 16. Recovery_R method demonstrated the 5-15% better performance rate than Recovery_S and the 7-20% better results than Recovery_F. In Fig. 3.2, we fix $m = 4$ and vary the values for n from 24 to 64. For this case, the Recovery_R algorithm was 10-20% faster than Recovery_S and 20-45% faster than Recovery_F.

XI. Performance of Silent Data Corruption Detection

The measurement of the performance of the SDC detection algorithm was performed within the methodology similar to that of data recovery outlined in the previous section. The stripe was reserved in the RAM, but this time the algorithm is aimed at finding the maximum possible number of the SDCs.

The red line shows the rate of the SDC detection (or their absence), while the blue line reflects the timing for finding all the SDC positions. Such separation is advisable because presented calculation module is focused on maximum productivity in normal mode and in case of errors. We consider two modes of calculation module:

Mode A. Calculation module determines only the fact of the SDC presence.

Mode B. Calculation module detects all the positions of SDCs and recovers them. This is the rare case with additional computational cost.

Fig. 4.1 corresponds to the case of the fixed $n = 128$ and changing values m from 2 to 16. Fig. 4.2 for a fixed $m = 4$ changing values n from 24 to 64. Red line illuminates the speed of the SDC presence detection, while the blue line reflects the speed of finding locations of all the SDCs.

XII. Further Optimization of Coding and Decoding

Since one can consider the stripe as matrix with number of rows equal to the number of blocks in the stripe and number of columns equal to the width of stripe, the encoding and decoding procedures described in Sections VI and VII are based on the matrix multiplication operation. One may expect significant optimization of both procedures on exploring the efficient matrix multiplication algorithms, such as, for instance, Strassen algorithm or modified Strassen algorithm [13].

This algorithm recursively reduces the multiplication of the initial matrices to that of their submatrices of the halved orders. Being applied to the data recovery procedure based on formula (13) with the stripe width of 4096 bytes, this has demonstrated the 10% optimization for the Recovery_R and Recovery_S algorithms. As for the Forney algorithm, the acceleration here is less considerable, due to the fact that the algorithm involves lesser matrix multiplications.

XIII. Conclusions

The paper provides a method for encoding and decoding of the RS codes with the ability to find and fix several silent data corruptions, even in the presence of failed blocks. The proposed method consists in the matrix representation of the both operations, namely, coding matrix for checksum calculation and recovery matrix for the error corrections. It provides also some improvements to the Berlekamp-Massey algorithm implementation, using SIMD instructions and procedures for fast multiplication by field elements on CPU registers. This approach is remarkable due to no additional memory required for storing intermediate calculations, has a

Fig. 3.1 Recovery of maximum possible amount of failed drives. Corresponds to the case of fixed n=128, and values of m varied from 2 to 16, stripe width=4096

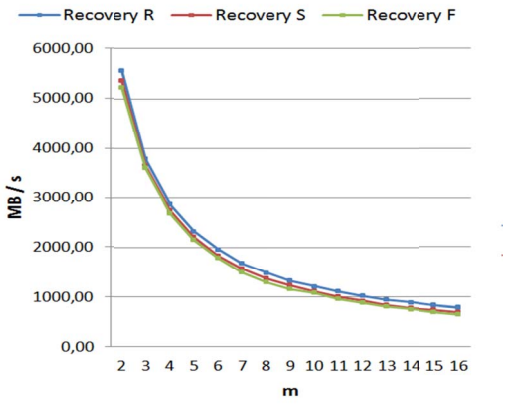


Fig. 3.2 Recovery of maximum possible amount of failed drives. Corresponds to the case of fixed m=4 and values of n varied from 24 to 64, stripe width=4096

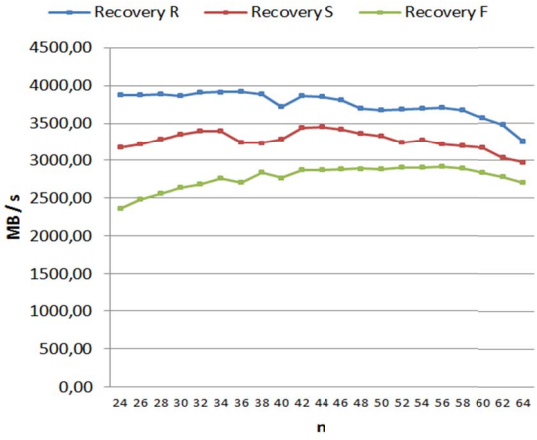


Fig. 4.1 Discovering and search of SDC n = 128, m = 2...16, the stripe width=4096

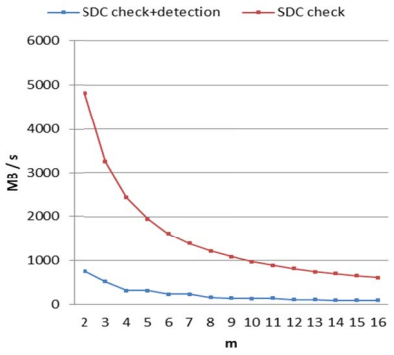
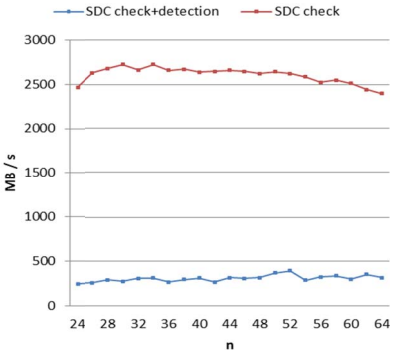


Fig. 4.2 Discovering and search of SDC n=24...64, m = 4, the stripe width = 4096



nice perspectives for parallelization, and algorithmic optimizations with the aid of the Strassen-like algorithms. We compare some variants of decoding aimed at optimizing the multiplication procedures in the Galois fields. It should be noted that though the proposed algorithms have been tested on a single computing core, they demonstrate a good performance rate. Therefore, one may expect that this rate will increase proportionally to the growth of the number of cores.

ACKNOWLEDGMENTS

The authors gratefully acknowledge the constructive suggestions of Prof Alexei Uteshev which helped to improve the quality of the paper.

REFERENCES

[1] Apache.org. "HDFS" http://hadoop.apache.org/docs/r1.2.1/hdfs_design.html

[2] S. Ghemawat, H. Gobioff, S.-T. Leung, "The Google file system" , Proceedings of SOSP 2003

[3] Inktank. "CEPH", http://ceph.com

[4] L. Pamies-Juarez, F. Oggier, A. Datta, “Decentralized Erasure Coding for Efficient Data Archival in Distributed Storage Systems”, ICDCN, 2013.

[5] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, S. Yekhanin, “Erasure coding in Windows Azure storage,” USENIX ATC, June 2012

[6] C. Huang, M. Chen, and J. Li, "Pyramid Codes: Flexible Schemes to Trade Space for Access Efficiency in Reliable Data Storage Systems," in Sixth IEEE International Symposium on Network Computing and Applications, 2007

[7] J. Plank, K. Greenan, E. Miller, Screaming Fast Galois Field Arithmetic Using Intel SIMD Instructions, FAST 2013: 11th USENIX Conference on File and Storage Technologies, San Jose, CA, February, 2013

[8] D. Ford, F. Labelle, F. Popovici, M. Stokely, V. A. Truong, L. Barroso, C. Grimes, S. Quinlan, “Availability in Globally Distributed Storage Systems." In The 9th USENIX conference on Operating Systems Design and Implementation (OSDI). 2010

[9] L. Bairavasundaram, G. Goodson, B. Schroeder, A. Arpaci-Dusseau, R. Arpaci-Dusseau, “An Analysis of Data Corruption in the Storage Stack”, TOS, 2008

[10] Tursin D. F., Platonov S. M., “Integrity data solutions in modern storage systems”, Herald of computer and Informational technologies, 2013

[11] H. Peter Anvin, “The mathematics of RAID-6”, https://www.kernel.org/pub/linux/kernel/people/hpa/raid6.pdf

[12] R. Lidl and H. Niederreiter. “Introduction to finite fields and their applications”, revised edition. Cambridge University Press, 1994

[13] T. Cormen, C. Leiserson, R. Rivest, C, Stein, “Introduction to algorithms” Third Edition, The MIT Press., p. 902, p.79-83.

[14] D. Knuth,” The Art of Computer Programming, Volume 1: Fundamental Algorithms”. p. 38.

[15] J. Plank “Erasure Codes for Storage Applications," Tutorial, FAST-2005: 11th USENIX Conference on File and Storage Technologies, San Jose, CA, February, 2005

[16] W. W. Peterson and E. J. Weldon, Jr. “Error-Correcting Codes, Second Edition.” The MIT Press, Cambridge, Massachusetts, 1972.

[17] E. Berlekamp, “Algebraic Coding Theory”, Laguna Hills, CA: Aegean Park Press.

[18] J. Plank, M. Blaum, “Sector-Disk (SD) Erasure Codes for Mixed Failure Modes in RAID Systems," ACM Transactions on Storage, Volume 10, Issue 1, January, 2014.

[19] Intel Intrinsics Guide, https://software.intel.com/sites/landingpage/IntrinsicsGuide/

[20] F. J. MacWilliams, N. J. Sloane. “The Theory of Error-Correcting Codes, Part I.” North-Holland Publishing Company, Amsterdam, New York, Oxford, 1977.

[21] E. A. Kruk, “Lekcii po Teorii kodirovaniya”, Saint-Petersburg State University of Aerospace Instrumentation (SUAI) .

[22] A. Soro, J. Lacan, “FNT-based Reed-Solomon Erasure Codes.”, 2010 7th IEEE Consumer Communications and Networking Conference.

[23] G. Jr. Forney, "On Decoding BCH Codes", IEEE Transactions on Information Theory.