A Framework for Coded Computation

Eric Rachlin and John E. Savage
Computer Science, Brown University
Providence, RI 02912-1910

***Abstract*— Error-correcting codes have been very successful in protecting against errors in data transmission. Computing on encoded data, however, has proved more difficult. In this paper we extend a framework introduced by Spielman [14] for computing on encoded data. This new formulation offers significantly more design flexibility, reduced overhead, and simplicity. It allows for a larger variety of codes to be used in computation and makes explicit conditions on codes that are compatible with computation. We also provide a lower bound on the overhead required for a single step of coded computation.**

I. INTRODUCTION

When symbols are transmitted across a noisy channel, one of the most basic approaches for protecting against errors is to use a repetition code. Since the birth of coding theory in the 1940's, however, many far more efficient codes have been discovered. Unfortunately, analogous results have not been obtained for noisy computation.

In this paper, we consider networks of noisy computing elements, logic gates, for example, in which each element can "fail" independently at random with probability ϵ. When an element fails, it outputs an incorrect value.

In 1956, von Neumann proposed the first systematic approach to building logic circuits from noisy gates [1]. His approach was to repeat each gate r times, then periodically suppress errors by taking many random majorities. This is very similar to protecting transmitted data using a repetition code. The main complication is that the majority gates can themselves fail. Thus, the new goal is to avoid error accumulattion while keeping the size of majority gates constant.

Thirty years later Pippenger successfully analyzed von Neumann's construction [2]. He demonstrated that given a fault-free circuit of size C, a fault-tolerant version of it, C', could be constructed of size $O(|C| \log |C|)$ such that for all inputs the probability that an output of C' is in error is within $O(\epsilon)$. An excellent description of this analysis can be found in [3]. Unfortunately the analysis also suggests that the constant associated with the $O(\log |C|)$ bound is large, as do experimental results [4].

After Pippenger obtained an upper bound on r, he and others obtained lower bounds for the size of von Neumann fault-tolerant circuits. Under the assumption that all gates fail independently with probability ϵ, the size and depth required for repetition-based fault tolerance has been shown to be within a constant factor of optimal for many basic functions (XOR for example) [5], [6], [7]. The derivation of these bounds highlight a shortcoming of the von Neumann model. Since all gates can fail with probability ϵ, the inputs and outputs of a circuit always have probability ϵ of being incorrect. For a sensitive function of N inputs, XOR for instance, each input must be sampled $O(\log N)$ times simply to ensure that information about its correct value reaches the output with high probability. In other words, since the inputs to a circuit are essentially encoded using repetition, the amount of redundancy for a reliable encoding is $\Omega(\log N) = \Omega(\log |C|)$.

This research was funded in part by NSF Grants CCF-0403674 and CCF-0726794.

In this paper, we consider a more general, and more realistic model of noisy computation in which some gates can be larger, but highly reliable (much like today's CMOS gates), while most gates are small, but susceptible to transient failures (an anticipated characteristic of nanoscale technologies [8]).

In this new model, most computational steps are done using noisy gates interspersed by a few steps in which reliable gates are used to decode and re-encode data without errors. This model more closely parallels data being transmitted over a noisy channel using a reliable encoder and decoder.

As we demonstrate, this **coded computation** model introduces a wide range of new design possibilities including new codes. Although this model has not been extensively studied, lower bounds on circuit size have been obtained. We will review and generalize these bounds.

II. RELATED WORK

Linear error correcting codes have been in use since the 1950s [9]. These are codes defined over finite fields in which the check symbols are linear combinations of the information symbols. When the computations that need to be coded are linear, it is known how to compute on such encoded data [10]. The problem is much more complex when the computations are non-linear. Early work in coded computation established simple lower bounds [11], [12], [13] (see Section IV) that suggest the difficulty of this problem.

Spielman [14] has proposed a general purpose approach to coded computation that we extend in this paper. First, he proposed that codes be used over alphabets that are supersets of the source data alphabets. These codes may use symbols from a larger alphabet for check symbols. Second, he proposed that the definition of functions over the smaller alphabet be extended to functions over the larger alphabet using interpolation polynomials. Third, he proposed that data be encoded using 2D Reed-Solomon (RS) codes.

In Spielman's approach, the result of computing on RS encoded data is RS encoded data using an RS code with smaller error correcting capability than the original code. This overcomes the lower bounds referenced above. However, it necessitates that the newly computed data be "transcoded" back to the original code so that another computational step can be taken on data in the original format. Both the computations and transcodings are done in a noisy environment.

The overhead of the Spielman approach is quite large. RS codes, along with his use of processor-based hypercube networks both introduce significant overhead. In our framework we demonstrate that this overhead can be reduced. More importantly, we describe how other codes and networks topologies can be employed. Unlike Spielman's approach, we offer a much wider range of design possibilities.

III. Coded Computation

In our general model of coded computation data is encoded, then computations are performed on data producing results in a potentially different code, which are mapped back to the original code.

A. Model of Computation

Our model of computation is that of a network of basic computing elements. These elements can be logic gates that compute simple Boolean functions, or more complex computing elements, such as full fledged processors computing arithmetic and logic functions, such as found in hypercubes.

In this model, a single step of computation on input vectors $\boldsymbol{x} = (x_1, x_2, \ldots, x_k)$ and $\boldsymbol{y} = (y_1, y_2, \ldots, y_k)$ is simply the application of functions from a set $H = \{\phi_j : F^2 \mapsto F \mid 1 \leq j \leq h\}$ to pairs of inputs. Let $\boldsymbol{w} = (w_1, w_2, \ldots, w_k)$ (instructions) identify the functions applied. Then, we denote this operation as $\boldsymbol{\kappa^{(k)}}$ and where

$$\boldsymbol{\kappa^{(k)}}(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{w}) = (\kappa(x_1, y_1, w_1), \ldots, \kappa(x_k, y_k, w_k))$$

and $\kappa(x, y, w) = \phi_w(x, y)$. When this model is applied to circuits, $F = \{0, 1\}$ and H may contain the functions $\phi_1(x, y) = \text{NAND}(x, y)$, and $\phi_2(x, y) = x$.

To perform multiple steps of computation, the output of $\mathbf{z}_t = \boldsymbol{\kappa^{(k)}}(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{w})$ is copied and each copy is permuted. At the next step, new instructions $\boldsymbol{w'}$ are applied to permuted data, as shown below, to compute the next value.

$$\mathbf{z}_{t+1} = \boldsymbol{\kappa^{(k)}}(\pi_1(\boldsymbol{z_t}), \pi_2(\boldsymbol{z}), \boldsymbol{w'})$$

Our computational model allows for π_1, π_2, and $\boldsymbol{w}$ to vary from step to step. The number of computations per step can also vary, but in this paper we assume it is constant.

We note that computations by logic circuits and hypercubes in which data is shared between processors that are adjacent along one dimension of the hypercube fit this model. Circuits can be limited to fanout two and "levelized" so that data moves synchronously from one level to the next. In such circuits gates have level l if the longest path to an external input passes through $l - 1$ gates. A wire passing through a level is augmented by introducing a "buffer" element at that level, thereby ensuring that wires only pass between levels.

Large reliable gates can be used to periodically decode and re-encode data so that the probability of a transcoding error is kept small.

B. One-Step of Coded Computation

Since the goal is to compute on encoded data and produce encoded data as a result, for this treatment we assume that systematic codes are used, although it is not strictly necessary. Consequently, when there are no component failures, the results of a computation in the absence of coding appear explicitly in the encoded result. The challenge is to find a way to combine the check symbols so that the results of a computation in the absence of failures is a codeword in a code with good error correction capability.

Let the inputs $\boldsymbol{x}$ and $\boldsymbol{y}$ to a step be encoded by a code C with encoding function $E : F^k \mapsto G^n$. That is, G is the codeword alphabet. G is assumed to be a superset of the input alphabet F, that is, $F \subseteq G$. Because we wish the coded results to be systematic, that is, to contain the k unencoded results of each step in the output codeword, we use the operator $\boldsymbol{\kappa^{(k)}}(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{w})$ to combine these values.

This formulation leaves unresolved how to combine the check symbols in the two codewords. For computations on hypercubes Spielman [14] proposes extending the definition of the function $\kappa(x, y, w)$ using interpolation polynomials, Φ. We refer to these as **extension polynomials**, one example of which is the interpolation of $\kappa(x, y, w)$ defined below. As we illustrate in Section III-E, extension polynomials need not be limited to interpolations over three variables.

An **interpolation polynomial** $\Phi(r, s, t)$ is given below in terms of $M(r, u, X) = \prod_{\rho \neq u, \rho \in X} \frac{r-\rho}{u-\rho}$, a function with value 1 when $r = u$ and 0 for $r \neq u$ when $r \in X$.

$$\Phi(r, s, t) = \sum_{u,v,w} \kappa(u, v, w) M(r, u, U) M(s, v, V) M(t, w, V)$$

Here $u \in U$, $v \in V$, and $w \in W$ where it is assumed that $U, V, W \subseteq F$. It follows that $\Phi(r, s, t) = \kappa(r, s, t)$ when r, s, and t are in U, V and W. However, $\Phi(r, s, t)$ is also defined for r, s, and t in G. The degrees of $\Phi(r, s, t)$ in r, s and t are $|U| - 1$, $|V| - 1$, and $|W| - 1$, respectively.

One step of encoded computation, denoted $E^*(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{w}) = \Phi^{(n)}(E(\boldsymbol{x}), \mathrm{E}(\boldsymbol{y}), E(\boldsymbol{w}))$, is defined to be the component-wise application of $\Phi(r, s, t)$ to the encodings $E(\boldsymbol{x})$, $E(\boldsymbol{y})$ and $E(\boldsymbol{w})$ of $\boldsymbol{x}$, $\boldsymbol{y}$ and $\boldsymbol{w}$ with code C.

$$\Phi^{(n)}(\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}) = (\Phi(\boldsymbol{a}_1, \boldsymbol{b}_1, \boldsymbol{c}_1), \ldots, \Phi(\boldsymbol{a}_n, \boldsymbol{b}_n, \boldsymbol{c}_n))$$

$E^*(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{w})$ is a codeword in a code C^* potentially different from C. Fault-tolerant computation requires that C^* has good error correcting capabilities. Such codes are discussed below.

Transcoding is the task of correcting errors in $E^*(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{w})$ produced by one step of computation and mapping the result to a word close to $E(\boldsymbol{\kappa^{(k)}}(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{w}))$. Transcoding is simplest if C and, thus C^*, are systematic. However, transcoding can be performed on nonsystematic codes as well. When C is systematic, $E^*(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{w})$ can be decoded (or simply corrected) to directly obtain the information symbols that correspond to $\boldsymbol{\kappa^{(k)}}(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{w})$. In the absence of errors, these symbols can then be re-encoded in C. However, in a noisy computation, this approach is not fault-tolerant, since a single decoding error could corrupt the entire transcoding process.

To provide fault-tolerance, Spielman proposes using 2D RS codes C. Such codes are frequently used to cope with burst errors. We observe that Spielman's basic approach can be made to work with any multidimensional linear code. In the case of an arbitrary systematic 2D linear code, for example, codes can be transcoded first by rows, then by columns. When transcoding is done one dimension at a time, an error that corrupts an entire row or column can still be corrected by transcoding along the other dimension. This works even when a different linear code is used in each dimension.

C. Codeword Permutations

Our model of computation requires that outputs from one stage of a computation be copied and permuted to supply inputs to the next stage. In a coded computation this means that codewords must be permuted.

If a permutation applied to a codeword doesn't return a codeword in the same code, it will be difficult to determine the nature of the code that results from the next computation step. Thus, it is essential that we be able to decompose arbitrary permutations into a series of individual permutations that do not change the code. In this section, we describe how a small set of allowable code-preserving permutations can be composed to perform arbitrary permutations.

1) Permuting Codewords Directly: Spielman [14] has observed that when $G = GF(2^q)$, RS codes are closed under permutations that correspond to data movement along a single dimension of a hypercube. RS codes, as well as many other codes, are also closed under cyclic shifts. Both types of permutation are sufficiently powerful to implement arbitrary permutations when combined with the correct choice of $\Phi(r, s, t)$.

Many network topologies exist which implement arbitrary permutations using a limited set of of permutations. For example, the permutations obtained by moving data along a common dimension of the hypercube can be composed to sort data using a Beneš network of back-to-back FFTs [15] where each node implements only the switching operation $\Phi(r, s, t) = rt + s(1-t)$ for $t \in \{0, 1\}$.

If permutations π_1 and π_2 are hypercube permutations, one can simulate data movement through a butterfly graph (FFTs). If π_1 and π_2 are one-unit cyclic shifts, they can simulate data movement on a shuffle exchange graph. This in turn can be used to simulate a butterfly graph, which allows for arbitrary data permutations [16].

2) Permuting during Transcoding: When 2D linear codes are used, arbitrary permutations of either rows or columns can be realized during transcoding. Recall that transcoding involves decoding and re-encoding data in each row followed by that in each column. After just the rows (or columns) have been decoded, however, the encoded columns (or rows) remain. These encoded columns (or rows) can be reordered before re-encoding takes place. This operation is sufficiently powerful to simulate hypercube-style data movement and, as a result, allows us to consider linear codes that are not closed under a specific set of permutations.

This important observation also allows us to use codes that don't allow for the direct application of $\Phi^{(n)}$ described in Section III-B. Instead of applying $\Phi^{(n)}$ directly to the encoded data, we can first decode the rows, apply $\Phi^{(n)}$ to the encoded columns, then re-encode the rows. We still apply $\Phi^{(n)}$ directly to encoded columns, but we avoid applying it to encoded rows. As a result, the row code can be any linear code we choose.

D. Comparison with Exisiting Models

The above model for coded computation captures the essence of the von Neumann [1] and Spielman [14] models. In the von Neumann model data at gates and gates themselves are replicated r times and the results computed. r majority gates are introduced, one for each copy of the original gate. Their inputs are pseudo-randomly selected subsets of the outputs of gate copies. These gates help to avoid error accumulation [2].

Spielman's model is the d-dimensional hypercube in which each processor receives an independent instruction stream and processors alternate between communication with their nieghbor(s) and performing local computations. Computations on other parallel machine models can be mapped to the hypercube efficiently. Spielman assumes that data moves between processors along one dimension of the hypercube on each time step. The operation ϕ_i performed by a processor is determined by its instruction and the processor design.

E. Examples of Extension Polynomials

If coded computation is to be done with a circuit and each of the gate operations is a NAND, the instruction word is constant and the input alphabet is $F = GF(2)$. Thus, the extension polynomial, which doesn't depend on the instruction, is $\Phi(r, s) = (1 - rs)$. Over $GF(2)$ it has the same value as NAND. If the gates consist of both NAND and buffer gates, the extension polynomial $\Phi(r, s, z) = (1 - rs)z + r(1 - z)$ returns the value of either the NAND of the two inputs when the instruction is $z = 1$ or the first of the two inputs when it is $z = 0$ and r and s are variables over $GF(2)$.

Extension polynomials need not have one variable for each input and instruction. For example, when the basis Ω of Boolean operations includes AND, OR and NOT, the extension polynomial $\Phi(r, s, z_1, z_2) = rsz_1z_2 + (r+s+rs)z_1(1-z_2) + (1-r)(1-z_1)(1-z_2)$ returns AND when $z_1 = z_2 = 1$, OR when $z_1 = 1$ and $z_2 = 0$, and NOT on r when $z_1 = z_2 = 0$, where addition and multiplication are over $GF(2)$, that is, addition is XOR and multiplication is AND. This polynomial allows binary codes to be used instead of requiring that codes be over large fields.

If coded computation is done with hypercubes, the extension polynomial depends on the processors employed. Complex processors may require very high degree extension polynomials. However, high degree polynomials can be implemented through successive applications of low degree polynomials, thereby minimizing their overhead.

F. Encoding with Linear Codes

The result of composing an extension polynomial $\Phi(r, s, t)$ component-wise with codewords $E(\boldsymbol{x})$, $E(\boldsymbol{y})$, and $E(\boldsymbol{w})$ in C generates a word in a new code C^* denoted $E^* : G^{3k} \mapsto G^n$. Our objective is to determine the properties of the code C^*.

Let the code C be linear with generator matrix M. Then, $E(\boldsymbol{x}) = \boldsymbol{x}M$, $E(\boldsymbol{y}) = \boldsymbol{y}M$, and $E(\boldsymbol{w}) = \boldsymbol{w}M$ where M is a $k \times n$ generator matrix over G and $\boldsymbol{x}$ is in F^k. Let $B_0 = \{\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_k\}$ be the **basis** $\boldsymbol{n}$**-vectors** of matrix M. Then,

$$E(\boldsymbol{x}) = \sum_{i=1}^{k} x_i \boldsymbol{b}_i \quad \text{and} \quad E(\boldsymbol{x})_j = \sum_{i=1}^{k} x_i b_{i,j}$$

When $\Phi(r, s, t)$ is applied component-wise to the encoded inputs, it produces the encoded output $E^*(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{w})$ whose jth component is given below.

$$E^*(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{w})_j = \Phi(\sum_{i=1}^{k} x_i b_{i,j}, \sum_{i=1}^{k} y_i b_{i,j}, \sum_{i=1}^{k} w_i b_{i,j})$$

Since $\Phi(r, s, t)$ is a multivariate polynomial, it contains products of powers of r, s, and t plus a constant term. Let $\boldsymbol{a} \wedge \boldsymbol{b} = (a_1 * b_1, a_2 * b_2, \ldots, a_m * b_m)$ denote the **parallel product** of basis vectors where $a_i * b_i$ is multiplication in the field G. Then, the output codeword C^* can be expressed as a linear combination of the parallel product of powers of basis vectors in B_0. The maximum degree of these products is the sum of the degrees of $\Phi(r, s, t)$ in r, s, and t.

G. Polynomial Input Codes

Spielman [14] has proposed using two-dimensional Reed-Solomon (RS) codes as the basic code C. Before introducing these codes, we consider one-dimensional RS codes.

An $[n, k, d]_q$ code has block length n, k information symbols, minimum distance d, and code alphabet of size q. A standard $[n, k, d]_{|G|}$ RS code is defined by polynomials. Given the k source values $\boldsymbol{a} = (a_0, a_1, \ldots, a_{k-1})$ from the finite field G the corresponding codeword $(c_0, c_1, \ldots, c_{n-1})$ is obtained by evaluating the polynomial $p_{\boldsymbol{a}}(u) = a_0 + a_1 u + \cdots + a_{k-1} u^{k-1}$ at n elements of G where $k < n$. That is, $c_j = p_{\boldsymbol{a}}(g_j)$ where g_1, g_2, ..., g_k are elements of the field G. It is easy to show that RS codes have minimum distance $d = n - k + 1$.

The extended RS codes are defined by the evaluation of one-dimensional interpolation polynomials. The interpolation polynomial $\tilde{p}_{\boldsymbol{a}}(u)$ is defined to have value a_j on g_j, that is, $a_j = \tilde{p}_{\boldsymbol{a}}(g_j)$. The extended RS codes are systematic.

The result of a coded computation using the extension polynomial $\Phi(r, s, t)$ is the word obtained by evaluating the polynomial $\Phi(\tilde{p}_{\boldsymbol{x}}(u), \tilde{p}_{\boldsymbol{y}}(u), \tilde{p}_{\boldsymbol{w}}(u))$ at n elements of G. The resultant words are codewords in an extended RS code C^* defined by a polynomial of degree $(|U|+|V|+|W|-3)(k-1)$ of length n. Thus, the minimum distance of this code is $d^* = n - (|U| + |V| + |W| - 3)(k - 1) + 1$ and the resultant code is an $[|G|, 3(k - 1), d^*]_{|G|}$ code. Clearly, the code has error correction capability only if $d^* \geq 3$ or $k \leq (n - 2)/(|U| + |V| + |W| - 3)$.

A 2D RS code can be defined in a similar way, except by evaluating a polynomial p of two variables at elements of G^2. A simpler approach, however, is to define these codes as one would define any 2D linear code. This is a code in which information symbols are arranged in a grid, and rows then columns are encoded. Since the code is linear, encoding the columns first, then the rows, will also produce the same code. It is also acceptable if the rows and columns are encoded using different linear codes.

As an alternative to RS codes one can employ Reed-Muller (RM) codes. Like RS codes, RM codes are also defined by evaluating a polynomial. Instead of a one-dimensional, degree-d polynomial, $p_{\boldsymbol{a}}(u)$, over an arbitrary field G, (r, m)-RM codes are formed by evaluating a r-dimensional, degree-m polynomial, $p_{\boldsymbol{a}}(u_1, ..., u_r)$ over $GF(2)^r$. Like RS codes, these codes are linear since each degree d polynomial is of the form $p_{\boldsymbol{a}}(u_1, ..., u_r) = \sum a_i p_i(u_1, ..., u_r)$, where each p_i is the product of at most m variables. The codeword corresponding to $p_{\boldsymbol{a}}(u_1, ..., u_r)$ is the value of the polynomial at all points in $GF(2)^r$. The minimum distance of this codes is 2^{r-m} [17].

Similar to RS codes, RM codes allow for applications of a polynomial Φ. For example, when the parallel product is taken of two codewords in an (r, m)-RM code it yields an $(r, 2m)$-RM code. Unlike RS codes, RM codes allow Φ to be realized as bitwise operations. Both classes of codes are closed under a wide range of permutations, for example, those corresponding to data movement on a hypercube.

H. Families of Parallel Product Codes

RS and RM codes allow us to apply a polynomial function to each position of codewords. Other families of codes based on polynomials allow for this operation as well. By performing parallel multiplication of basis vectors, we can construct many families of linear codes that meet this requirement. We present our construction below, then highlight its connection to existing codes.

1) To define a family of codes, begin with an initial linear code C_0 over a field F. Since C_0 is linear, each codeword is a linear combination of basis vectors B_0.
2) Let $\wedge(B_0)$ be the set of all possible parallel products of pairs of basis vectors in B_0.
3) We use the parallel product operation to construct a family of codes. Let $B_i = \wedge(B_{i-1}) \bigcup B_{i-1}$ and let C_i be the code consisting of all linear combinations of vectors in B_i.

It is straightforward to show that $|B_i| \leq (|B_0| + 1)^i$. It is also straightforward to show that if a codewords belongs to C_i, then application of a degree m polynomial, $\Phi(r, s, t)$, results in a codeword in C_{i+m-1}. This result applies to extension polynomials in more than three variables.

IV. Lower Bounds

Given how successful error correcting codes have been against data transmission errors, their use in computation seems natural. Some early lower bounds suggested that this was difficult, and that under somewhat restrictive assumptions, one cannot do better than simple gate repetition.

In a single step $E^*(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{w}) = \boldsymbol{\Phi^{(n)}}(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{w})$ is computed. If for each value of j, w_j denotes $\oplus$ (XOR), addition over $GF(2)$, the computation is easily encoded by a linear error correcting code over $GF(2)$ with encoding function E_L. Then $E^*(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{w}) = E_L(\boldsymbol{x}) \oplus E_L(\boldsymbol{y})$ where $\oplus$ denotes vector XOR.

In this case C^* and C are the same and a subsequent transcoding step is unnecessary. This holds true if each instruction is the XOR of the complement of one or both of its inputs.

Consider computations in which each operation is of AND-type, that is, it is the AND with complementation on one or both inputs or the output. When the codes C and C^* are the same, the following holds.

Theorem IV.1 ([13]) *Let C be an $[n,k,d]_2$ code. When two codewords in C are combined component-wise with an AND-type Boolean operation and the resultant code $C^* = C$, then $n \geq kd$. That is, C is no more efficient than simply repeating each symbol d times.*

This lower bound demonstrates that you can't do better than a repetition code when $\Phi(r,s,t)$ is applied with no transcoding. It does not, however, take into account the possibility of applying a function of more than one symbol per codeword. The next theorem considers the case which the code alphabet is non-binary and each output can be a function of up to c components of each codeword. The result shows that the lower bound on n of Theorem IV.1 can only be decreased by increasing c or $|G|$.

Theorem IV.2 *Let C be an $[n,k,d]_{|G|}$ code and let $E : G^n \mapsto G^n$ be its encoding function. Let $T : G^{2n} \mapsto G^n$ compute the encoding of the component-wise AND of two k-tuples from the encodings of these two tuples. That is, $T(E(\boldsymbol{x}), E(\boldsymbol{y})) = E(\boldsymbol{x} \wedge \boldsymbol{y})$. If each output of T, T_i, is a function of at most c symbols of $E(\boldsymbol{x})$ and $E(\boldsymbol{y})$), the inequality $n \geq kd/(c \log_2 |G|)$ must hold.*

Proof: Let $T(\boldsymbol{\alpha}, \boldsymbol{\beta})_i$ denote the ith component of $T(\boldsymbol{\alpha}, \boldsymbol{\beta})$. By assumption $T(\boldsymbol{\alpha}, \boldsymbol{\beta})_i$ depends on at most c components of $\boldsymbol{\alpha}$ and of $\boldsymbol{\beta}$. Let $S(i)$ denote these components of $\boldsymbol{\alpha}$. Then, $|S(i)| \leq c$.

Let $\mathbf{1}_r$ be a k-tuple in which all components are 0 except for the rth which has value 1. Also, let $\mathbf{0}$ be the k-tuple in which all components are 0.

By definition, $T(E(\boldsymbol{x}), E(\mathbf{1}_r)) = E(\mathbf{1}_r)$ or $E(\mathbf{0})$ depending on whether $x_r = 1$ or $x_r = 0$.

Let $\zeta_i^r(E(\boldsymbol{x})) = T(E(\boldsymbol{x}), E(\mathbf{1}_r))_i$ denote the ith component of $T(E(\boldsymbol{x}), E(\mathbf{1}_r))$. $\zeta_i^r(E(\boldsymbol{x}))$ depends on $|S(i)|$ components of $\boldsymbol{x}$. By assumption, $|S(i)| \leq c$.

Because the code has minimum distance d, there are at least d positions at which the codewords $E(\mathbf{1}_{\boldsymbol{r}})$ and $E(\mathbf{0})$ differ. Let $I(\mathbf{0}, r)$ denote these positions. Observe that for each r, we can select i in $I(\mathbf{0}, r)$ and compute $\zeta_i^r(E(\boldsymbol{x}))$. Since $\zeta_i^r(E(\boldsymbol{x})) = E_i(\mathbf{0})$ if and only if $x_r = 0$, knowing $\zeta_i^r(E(\boldsymbol{x}))$ reveals the value of x_r.

Let $E^{S(i)}(\boldsymbol{x})$ denote the components of $E(\boldsymbol{x})$ in positions $S(i)$, $|S(i)| \leq c$. It follows that $E^{S(i)}(\boldsymbol{x}) \in G^c$.

Let R_i denote the values of r such that $\boldsymbol{i} \in I(\mathbf{0}, r)$. Knowing $E^{S(i)}(\boldsymbol{x})$ reveals x_r for any $r \in R_i$. Since $E^{S(i)}(\boldsymbol{x})$ takes at most $|G|^c$ possible values and each variable x_r takes two values, $2^{R_i} \leq |G|^c$ or $|R_i| \leq c \log |G|$.

Pairs (r,i) satisfying $r \in R_i$ if and only if $i \in I(\mathbf{0}, r)$ are called **linked pairs**. The total number of linked pairs, Q, can be counted two ways:

$$Q = \sum_{r=1}^{k} |I(\mathbf{0}, r)| = \sum_{i=1}^{n} |R_i|$$

Since we know $|I(\mathbf{0}, r)| \geq d$, and $|R_i| \leq c \log_2 |G|$, we have $kd \leq cn \log_2 |G|$, the desired bound. ∎

The argument of this proof applies to any "AND-like" function such as NAND, OR, or NOR and to any Boolean functions of two or more variables that are "partially sensitive," meaning they can made independent as well as dependent on some variable by choice of values for the remaining variables.

V. Conclusions

We have outlined a framework for fault-tolerant coded computation. It extends an approach introduced by Spielman but offers significantly more design flexibility, reduced overhead, and simplicity. Coded computation has the potential to outperform the performance of the repetition-based model. Our approach is based on applying extension polynomials to parallel product codes and simulating the data movement steps required for arbitrary computations by composing a fixed set of permutations permitted by the transcoding process.

References

[1] John von Neumann. Probabilistic logics and the synthesis of reliable organisms from unreliable componets. In C. E. Shannon and J. McCarthy, editors, *Automata Studies*, pages 43–98, 1956.

[2] Nicholas Pippenger. On networks of noisy gates. In *Procs. 26th IEEE FOCS Symposium*, pages 30–38, 1985.

[3] Peter Gacs. Reliable computation. Technical report, Department of Computer Science, Boston University, 2005.

[4] Jie Han, J. Gao, P. Jonker, Yan Qi, and J.A.B. Fortes. Toward hardware-redundant, fault-tolerant logic for nanoelectronics. *IEEE Design and Test of Computers*, 22(4):328–339, July-Aug. 2005.

[5] N. Pippenger, G.D. Stamoulis, and J.N. Tsitsiklis. On a lower bound for the redundancy of reliable networks with noisy gates. *Information Theory, IEEE Transactions on*, 37(3):639–643, May 1991.

[6] Gacs and Gal. Lower bounds for the complexity of reliable boolean circuits with noisy gates. *IEEETIT: IEEE Transactions on Information Theory*, 40, 1994.

[7] William S. Evans. Information theory and noisy computation. Technical Report TR-94-057, Berkeley, CA, 1994.

[8] André DeHon. Law of large numbers system design. In Sandeep K. Shukla and R. Iris Bahar, editors, *Nano, Quantum and Molecular Computing: Implications to High Level Design and Validation*, pages 213–241. Kluwer, 2004.

[9] R. W. Hamming. Error detecting and error correcting codes. *Bell Syst. Techn. J.*, 29(2):147–160, 1950.

[10] C. Hadjicostis and G. Verghese. Fault-tolerant linear finite state machines, 1999.

[11] Peter Elias. Computation in the presence of noice. *IBM J. Res. Develop.*, 2:346–353, 1958.

[12] W. W. Peterson and M. O. Rabin. On codes for checking logical operations. *IBM Journal of Research and Development*, 3(2):163–168, 1959.

[13] S. Winograd. Coding for logical operations. *IBM Journal of Research and Development*, 6(4):430–436, 1962.

[14] Daniel A. Spielman. Highly fault-tolerant parallel computation. In *Procs. 37th IEEE FOCS Symposium*, pages 154–163, 1996.

[15] V. E. Beneš. Permutation groups, complexes, and rearrangeable multistage connecting networks. *Bell Syst. Techn. J.*, 43:1619–1640, 1964.

[16] H. S. Stone. Parallel processing with the perfect shuffle. *IEEE Trans. Computers*, C-20:153–161, 1971.

[17] Jacobus H. van Lint. *Coding Theory*. Springer-Verlag, Lecture Notes in Mathematics, Berlin, 1973.