Fault Tolerant Arithmetic with Applications in Nanotechnology based Systems

Wenjing Rao
UC San Diego
CSE Department
wrao@cs.ucsd.edu

Alex Orailoglu
UC San Diego
CSE Department
alex@cs.ucsd.edu

Ramesh Karri
Polytechnic University
ECE Department
ramesh@india.poly.edu

ABSTRACT

Several emerging nanotechnologies have been displaying the Negative Differential Resistance (NDR) characteristic, which makes them naturally support multi-valued logic with a large number of logic states. Such multi-valued logic with a large number of logic states can support a native digit-level redundant number system and hence a native digit-level carry save arithmetic. In this paper we present a new approach to linear block code based fault-tolerant arithmetic in NDR nanotechnologies. Specifically, we show how linear block codes can be used for error checking and error correction in carry save arithmetic operations. The proposed approach significantly improves timing and fault-tolerance of arithmetic operations in the highly unreliable nanoelectronic environment. Since digit-level information redundancy via linear block codes is widely used for fault tolerant communications and storage systems, the proposed scheme also unifies the fault tolerance approaches across arithmetic, interconnection and storage subsystems.

1. INTRODUCTION

Although a single nanotechnology cannot be anticipated to be dominant with certainty, a number of reasonable predictions can be made by studying the common features among them [1, 2]. One of the most important characteristics of all the nanotechnologies is the high unreliability of the constituent devices. In comparison to the 10^{-9} to 10^{-7} failure rates in CMOS technology, the failure rates in emerging nanotechnologies are projected to be in the order of 10^{-2} to 10^{-1} due to their size and frequency characteristics [2, 3]. These highly unreliable nanoelectronic devices elevate aggressive fault tolerance into an essential characteristic of a system design [1, 2, 4]. To ensure correct operation in the presence of permanent and transient faults, concurrent error detection, correction, diagnosis and reconfiguration need to be aggressively embedded in future nanotechnology based systems.

Hardware duplication and comparison is one approach to fault-tolerance. In this approach, error correction is achieved through a majority vote among multiple computations. Illustrative fault tolerance strategies of this type include R-fold module redundancy and NAND multiplexing [5]. Since large amounts of computational resources are available in nanotechnology based systems, hardware redundancy has been perceived as an attractive approach. NAND multiplexing [6], the Teramac reconfigurable system [7] and the embryonics system [8] constitute approaches in this direction. However, research has shown that, in order to achieve an acceptable reliability, enormous amounts of hardware redundancy (in the order of 10^3 to 10^5) might be necessary to cope with failure rates in the order of 10^{-2} to 10^{-1} [3].

Recomputation in time is an alternative approach to concurrent error detection and correction to guard against transient faults [9, 10]. Since the computation is performed a second time with the same unit, this approach requires less hardware overhead. However, time redundancy based fault tolerance delays the computation, thereby compromising system performance. Furthermore, it is not directly applicable to detect permanent faults.

Information redundancy is widely used to implement fault tolerance in communication and storage systems. Error detection and correction are performed by organizing the redundant information in the data according to a strict algebraic structure. Since arithmetic operations transform the input data, this strict algebraic structure is not always preserved. This is especially true for arithmetic operations in a binary logic system. For example, even in the basic addition operation, propagation of the carry bits precludes enforcement of a strict structure necessary to perform concurrent error detection and correction. In CMOS technology, several information redundancy based arithmetic fault-tolerance techniques have been developed [11, 12].

Carry save arithmetic is a popular logic level technique to speed up arithmetic operations such as addition, subtraction, constant multiplication and general multiplication with little or no area overhead [13]. The carry save technique speeds up an operation by using a redundant number system to eliminate carry propagation. Eliminating carry propagation provides an additional advantage in that it enables information redundancy based fault tolerance. However, CMOS technology which implements binary logic does not natively support a redundant number system. Hence, a redundant number system based carry save arithmetic in a binary system is typically implemented in an *ad hoc* fashion where extra binary bits are provided to artificially add redundancy.

1.1 Motivation

Notwithstanding its unsuitability in CMOS technology, in this paper we pursue information redundancy as a practical fault tolerance technique for arithmetic operations in emerging nanotechnologies based on the following observations:

- Negative Differential Resistance (NDR) is an important characteristic of RTD [14] and molecular electronics [15]. These NDR nanotechnologies naturally support multi-valued logic with large numbers of logic states. Since multi-valued logic with such a large number of logic states can support native digit-level redundant number systems, carry save arithmetic is ideal for these emerging NDR nanotechnologies. Such an efficient digit-level redundant number system can also effi-

 0-7803-8580-2/04 $20.00 Copyright 2004 IEEE

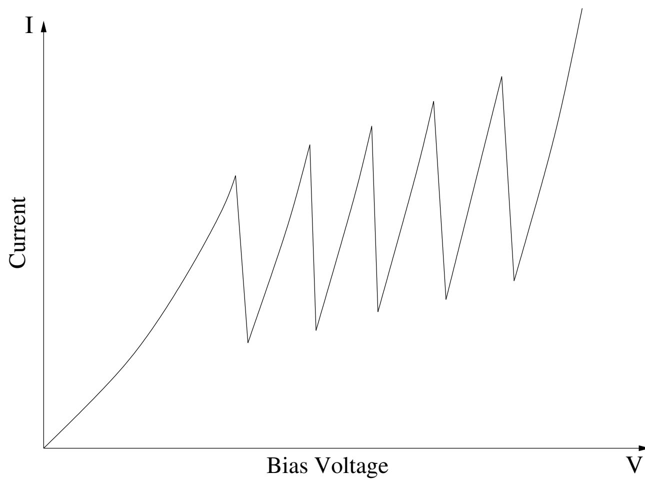

Figure 1: Multipeak characteristic of I-V curves of negative differential resistance nanotechnologies

ciently implement linear block code based carry save arithmetic for fault tolerance purposes. Recent research has explored the design of multi-valued logic based arithmetic [16] and binary logic based carry save addition using 1-out-of-3 codes [11].

- In emerging systems based on nanotechnologies, interconnection becomes dominant. Fault tolerance strategies will be needed to ensure reliable data transfer. In data communications theory, linear block codes have been used for reliable delivery of data between two points. These techniques can be adapted to support reliable data transfers over unreliable interconnections in nanotechnologies.
- A unified linear block code based fault tolerance approach to interconnection, storage and arithmetic subsystems can minimize and possibly eliminate encoding/decoding time and the associated hardware overhead.

In this paper we will show how the large number of logic states in NDR nanotechnologies can be efficiently used to support fault tolerance based on the theory of linear block codes. In Section 2 we review the multi-valued logic capability of NDR nanotechnologies, carry save arithmetic and the theory of linear block codes. In Section 3, we show first as a running example how a linear block code based fault tolerance scheme can be used for carry save arithmetic operators by using carry save addition. We then follow up this discussion with extensions to other arithmetic operations and summarize the overall fault tolerance design approach. In Section 4, we demonstrate the error detection and correction capabilities of the proposed scheme. Finally, Section 5 concludes the paper.

2. PRELIMINARIES

2.1 NDR based Multi-valued logic

Nanotechnologies such as resonant tunneling diodes (RTD) [14], resonant tunneling transistors (RTT) [14] and molecular electronics [15] have been investigated for their potential to enable ultra-high speed and extremely dense circuits. The current-voltage curves in these nanotechnologies display a large negative differential resistance (NDR) due to their physical characteristics. Figure 1 shows an example of the multipeak current-voltage curve with five peaks.

	5	8	10	7	11	Operand digit in [0, 11]
+	6	7	3	2	11	
	11	15	13	9	22	Position sums in [0, 22]
	1	5	3	9	2	Interim sums in [0, 9]
1	1	1	0	2		Transfer digits in [0, 2]
1	2	6	3	11	2	Sum digits in [0, 11]

Figure 2: Carry save addition supported by a 23-valued NDR nanotechnology.

The multiple-folded nature of the current-voltage curves in these NDR nanotechnologies facilitates compact multi-valued logic circuit implementations. An N-peak NDR device can have $N + 1$ states. Due to the high switching speed and the folded nature of the I-V curves, multi-valued logic with a large number of logic states has been shown to be of practical importance [17]. Fifteen state SRAM cells [17], six state counters [18] and multiple-state logic gates [19] have been designed.

Multi-valued logic implementations in CMOS technologies are impractical; the individual MVL building blocks require a large number of devices, and they also operate in the threshold mode, which results in poor operating speeds and noise margins [14].

2.2 Carry Save Arithmetic

The carry save technique has been developed to cope with the carry propagation delay problem in arithmetic unit design [20]. Carry save arithmetic uses a redundant number system to absorb the carry bits, thereby eliminating carry propagation. Thus carry save arithmetic operations can be performed with a constant delay.

Consider a redundant number system that can represent any number in the range [0, 2α]. Consider a carry save addition of two operands in radix r. If the operand digits are limited to [0, α], then the sum of any two operand digits falls in the range [0, 2α] and can be represented without any carry digit. Let us denote this sum digit in the range [0, 2α] as the *position sum*. The position sum needs to be translated back into the range [0, α] for any additional processing. The position sum is decomposed into an *interim sum* and a *transfer digit* where:

position sum = transfer digit × radix + interim sum

The interim sum digits are in the range [0, $r - 1$] and the transfer digits are in the range [0, $\lfloor(2\alpha)/(r-1)\rfloor$]. If

$$\lfloor(2\alpha)/(r-1)\rfloor + r - 1 \leq \alpha \quad (1)$$

then adding an interim sum digit to a transfer digit yields a final sum in the range [0, α] without generating any carry.

To clarify the general carry save principle, consider an example radix $r = 10$ carry save addition implemented using a 23-valued NDR nanotechnology as shown in figure 2. In order to eliminate carry generation and propagation, the operands in this example do not use the entire representation range supported by the 23-valued logic. Rather the operands are limited to the range [0, $\alpha = 11$]. Since inequality 1 holds, the position sum can be decomposed in such a way that all the digits in the final sum digits are always in the range [0, 11].

In binary logic (where $r = 2$) inequality 1 does not have positive solutions for α, hence making it infeasible to perform carry save addition on two operands. Instead, the carry save technique is only used to reduce the number of operands (typically from 3 to 2) of multi-operand additions. Three operands can be reduced by carry save addition into an interim sum and a transfer bits vector. Thus, carry save adders can be used as an important building block in binary multipliers.

On the other hand, since NDR nanotechnologies can implement multi-valued logic with a sizable number of logic states, they can support redundant number systems that satisfy inequality 1. The entire addition can then be performed without carry propagation. Specifically, the carry save technique can speed up the intermediate multi-operand addition as well as the final sum calculation. Furthermore, since multi-valued logic can directly represent the position sums, it can support linear block code based error detection and correction in all the steps of the carry save addition. Since the number of logic states in the multi-valued logic needs to support the range [0, $n \times \alpha$], the operand range α can be traded off with the number of operands n in a carry save addition.

2.3 Linear Block Codes

A linear block code $\mathcal{C}(n, k)$ encodes a k-digit[1] dataword into an n-digit codeword ($n > k$). The $(n-k)$ redundant digits are used to enforce the algebraic structure for each valid codeword [21]. The algebraic structure of a linear block code is based on a *finite field* $GF(q)$ of q elements. The field operations of addition, subtraction, multiplication and division can be performed on the field elements. A well constructed linear block code can achieve the maximum fault tolerance capability provided by the redundant information. For example, a (7, 4) Hamming code is a linear block code with $n = 7$ and $k = 4$ using three digits of redundant information to provide 2-digit error checking and 1-digit error-correction capability for any 4-digit dataword. Linear block codes have been widely applied in communication systems due to their well-defined structure and scalability: the level of error detection/correction capability can be precisely determined from the structure.

The algebraic property of a finite field $GF(q)$ requires that $q = p^m$, where p is a prime number and m is a positive integer. The number of field elements is either a prime number p (i.e. $m = 1$) or a power of a prime number p (i.e. $m > 1$). When $m = 1$, the field operations addition and multiplication are identical to arithmetic addition and multiplication modulo p.

A linear block code $\mathcal{C}$ is a subspace of the linear vector space $GF(q)^n$. Therefore, each element of $\mathcal{C}$ (i.e., an n-digit valid codeword) is a vector on the finite field $GF(q)$. Due to linearity, the linear combination of any two codewords is always a valid codeword. Since the zero vector is a member in any linear vector space, it is always a valid code. A codeword can be generated by multiplying a dataword with a *generator matrix* G. The generator matrix G can be represented in a *systematic form* of $G = [I|P]$, where I is the identity matrix and P is a $k \times (n-k)$ matrix. The resulting codeword has the original dataword as its leftmost k digits.

The Hamming distance of two codewords v_1 and v_2 is the number of digits that differ between v_1 and v_2. The Hamming weight of a codeword v is the number of non-zero digits in it. A basic lemma shows that, if d is the minimum Hamming weight over all non-zero codewords in $\mathcal{C}$, then the Hamming distance between any two codewords must be a multiple of d. The minimum Hamming distance of any two valid codewords is also d.

The set of vectors that span the vector space orthogonal to $\mathcal{C}$ constitutes the check matrix H for $\mathcal{C}$. Since H is the orthogonal complement of G, multiplying a valid codeword by H always generates a zero *syndrome vector*. A non-zero result, defined as a *syndrome vector*, indicates an invalid codeword and can identify the corresponding valid codeword from a syndrome table. Suppose an error occurs and changes e digits of a codeword c into c': if $e < d$, then the error is detectable since the error vector c' cannot be a valid codeword. If $e \leq \lfloor (d-1)/2 \rfloor$, then the error can be corrected as the codeword with the minimum Hamming distance to c'.

[1] We use *digit* instead of *bit* so as to indicate that the number system is based on multi-valued logic, rather than limited to binary logic.

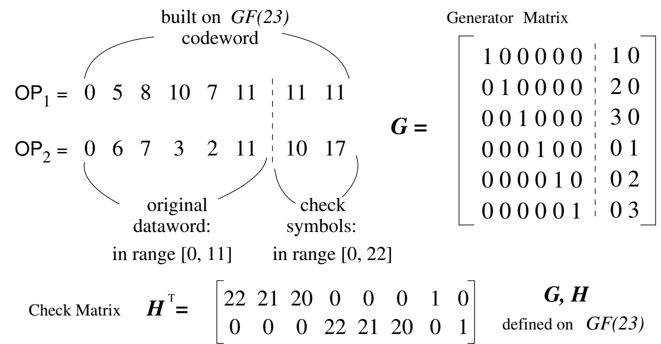

Figure 3: An example linear block code on $GF(23)$. A codeword is composed of four data digits and two check symbols.

3. FAULT TOLERANT CARRY SAVE ARITHMETIC

In this section we present linear block code based fault-tolerant carry save arithmetic. We first describe the error checking and error correction techniques for carry save addition with an example. We then show how these techniques can be extended to other arithmetic operations as well. Finally, we outline the general design flow for building such fault tolerant arithmetic operations.

3.1 Concurrent Error Checking in Carry Save Addition

Let us return to the running example of figure 2 that is based on an example NDR nanotechnology supporting twenty three logic states. It can natively support a redundant number system in the range [0, 22] based on the finite field $GF(23)$.[2] Since the final sum might be one digit longer than the operands, we extend the operands by adding a zero to the left.

Figure 3 shows the generator matrix G, the check matrix H and two codewords. The code symbols are generated by multiplying the six data digits with G. For example, for the data word OP_2, the first check symbol is obtained as $10 = (6 \times 2 + 7 \times 3) \bmod 23$ by multiplying it with the second to last column of G. Similarly, the second check symbol is obtained as $17 = (3 \times 1 + 2 \times 2 + 11 \times 3) \bmod 23$ by multiplying it with the last column of G. Since the generator matrix G is in a systematic form, the left six digits of the codeword are identical with the dataword and the rightmost two digits serve as the check symbols. From the linear block code theory, the validity of a codeword can be checked by multiplying it with the check matrix H and checking if the result is zero.

As can be seen from the example in figure 2, a carry save addition

[2] In order to use the algebraic structure to provide error checking, we need to choose a finite field with a prime number of elements to construct a linear block code. The prime number 23 serves therefore as a suitable illustration.

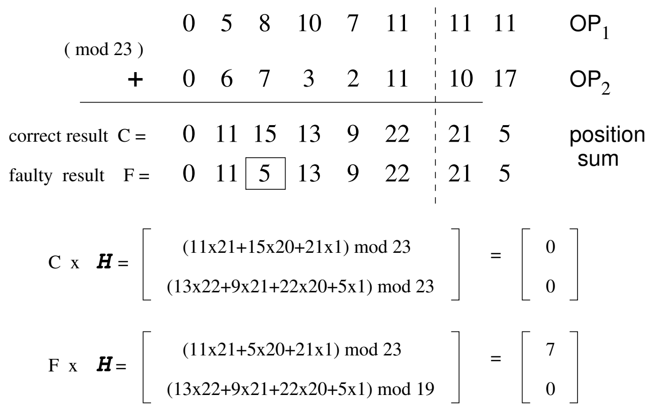

Figure 4: Error checking during position sum calculation entails multiplying the check matrix H with the position sum.

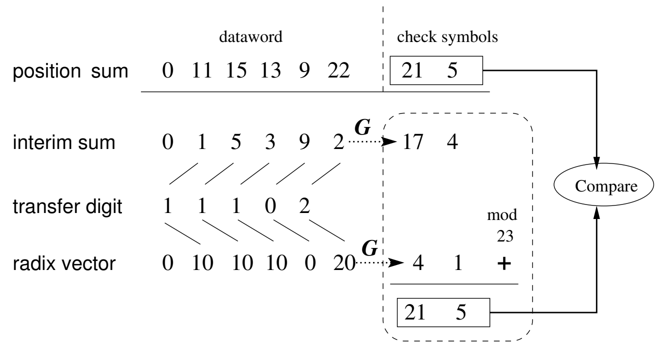

Figure 5: Error checking for the position sum decomposition stage entails checking if codewords for the transfer digits vector and the radix vector add up to the check symbols of the position sum.

is performed in three stages:

Stage 1: The position sum is calculated by adding the two operands digit by digit.

Stage 2: The position sum is decomposed into an interim sum and the transfer digits.

Stage 3: The interim sum digits and the shifted transfer digits are added to generate the final sum.

3.1.1 Error checking during the position sum calculation

Figure 4 explains the error checking during position sum calculation using the running example. The two 6-digit operands are encoded into 8-digit codewords. The two 8-digit codewords are added modulo 23 to generate an 8-digit position sum. The correctness of the position sum calculation can be checked by multiplying the position sum with the check matrix H and checking if the result is a zero vector.

Figure 4 also shows the situation when an error occurs in the third digit of the position sum and changes the correct value 15 to the faulty value 5. When the position sum is multiplied by the check matrix H, it yields a non-zero syndrome vector (7, 0), indicating an error. This syndrome vector can also be used to determine the correct result using a syndrome table [21].

3.1.2 Error checking the position sum decomposition

In the second stage of a carry save addition, the position sum is decomposed into the interim sum and transfer digits. This decomposition does not maintain the algebraic structure of the linear block code. Therefore, we cannot obtain a valid interim sum codeword and a valid transfer digit codeword.

However, the position sum is a linear combination of the interim sum and the transfer digits. Fault tolerance in this stage can be approached based on this observation. Figure 5 shows the error checking procedure. The transfer digit vector is shifted right by one digit and multiplied by the radix to yield a *radix vector*. The radix vector is in fact, a digit-by-digit difference between the position sum and the interim sum. Hence, the position sum is addition modulo 23 of the radix vector and the interim sum vector. The interim sum and radix vector are encoded by multiplying them with the generator matrix G. Error checking and error correction are then performed by checking whether adding the code symbols in the interim sum with the code symbols in the radix vector generates the corresponding code symbols in the position sum.

3.1.3 Error checking the final sum

The final sum is obtained by adding the interim sum with the transfer digits. Since each interim sum digit is in the range [0, 9] and each transfer digit is in the range [0, 2], every digit in the final sum is in the range [0, 11]. Since this addition is modulo 23, the algebraic structure of the linear block code is preserved. In a manner similar to Stage 1, faults in this stage can be detected by checking the validity of the final sum codewords. This error checking strategy is summarized in Figure 6.

3.2 Concurrent Error Correction

The error correction capability in the linear block codes is determined by the minimum Hamming distance d between every two codewords. When an error occurs with e digits changed from a valid codeword c, if $e \leq \lfloor (d-1)/2 \rfloor$, then the resultant error vector c' can be corrected to its nearest valid codeword. Typically, the implementation of error correction in the linear block code approach is performed by building up a syndrome table, which stores the valid codeword for each group of erroneous codewords. By looking up the resultant syndrome vector in the syndrome table, errors changing e digits with $e \leq \lfloor (d-1)/2 \rfloor$ can be corrected [21].

3.3 Extension to Other Arithmetic Operations

The proposed technique can be extended to other carry save arithmetic operations as follows:

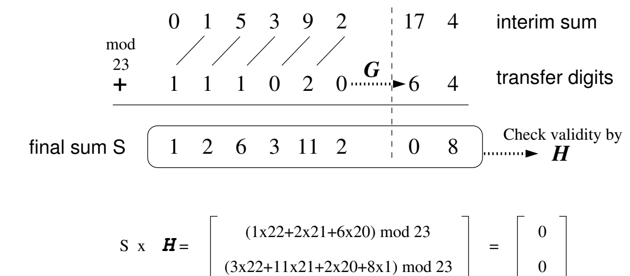

Figure 6: Error checking for the final sum entails multiplying the check matrix H with the final sum.

- Subtraction is equivalent to adding the complement of a number; therefore, the proposed scheme can be directly applied.

- Multiplication with constants is typically implemented as a collection of shifts-and-adds. Since the error detection and correction for the shift operation by linear block code is easy to implement, the proposed scheme can be applied too.

- A general multiplication process uses multi-operand additions, and is typically implemented by organizing several carry save addition blocks into a tree structure. Within each tree node, the carry save addition block can accomplish a constant delay for the multi-operand addition; thus the total delay of the multiplication is proportional to the depth of the tree, which equals the logarithm of the operand length.

  The carry save addition block in a multi-operand addition process performs a reduction on operand number by converting the operands into an interim sum vector and possibly multiple transfer digit vectors. This process is the same as the one in Stage 1 and Stage 2, defined in section 3.1. Therefore, the proposed approach can be applied to provide error detection/correction capability for these stages as was shown previously.

  In binary systems, the final sum needs to be calculated by carry propagation additions, since the number system cannot reduce the number of operands from two to one. However, under the multi-valued logic environment, the final sum can be also calculated with carry save addition as Stage 3 of the example in section 3.1. Therefore, carry propagation can be eliminated in the entire multiplication process; thus the proposed fault tolerance scheme not only can be applied, but also will benefit the general multiplication operation.

3.4 Putting it all together

First, a finite field $GF(q)$ is identified as a basis for the linear block code. The number of elements in the field, q, should be a prime number greater than the maximum possible value for the position sum digits. The arithmetic operations are then identical to operations modulo q. The largest value in the redundant number representation supported by the underlying nanotechnology should also exceed q.

The algebraic field structure of the resulting linear block code is used for error detection and correction during these arithmetic operations. Various levels of fault tolerance can be embedded into a system based on the expected defect rate by using an appropriate generator matrix G. Research in algebraic coding has provided methods to construct G to check and correct a specified rate of errors [21].

The check matrix H is generated as the orthogonal complement of G. If G is systematic, the resulting codeword always contains the original dataword as its prefix. To encode, a dataword vector is multiplied by G modulo q. Similarly, a codeword is checked by multiplying it with H modulo q and checking if it yields a zero syndrome vector. A non-zero syndrome vector can also be compared to a syndrome table for error correction.

Figure 7 shows this overall flow of incorporating fault tolerance into the carry save addition. In Stage 1, the two operands are encoded using G. The resulting operand codewords are added modulo q to form a position sum codeword. This stage is checked by validating the position sum codeword using H. In Stage 2, the position sum is decomposed into the interim sum and the transfer digits. In order to check the correctness of this procedure, the transfer digits

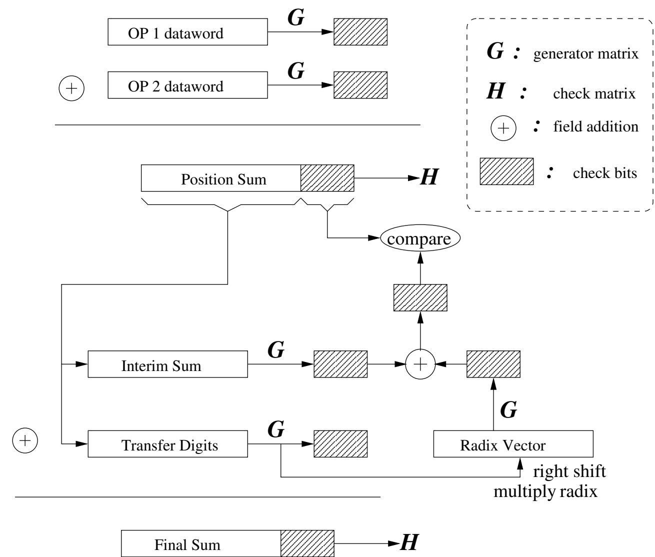

Figure 7: Fault tolerant carry save addition

are right-shifted one digit and multiplied by the radix to form the radix vector. The check symbols for the radix vector are added to the check symbols of the interim sum modulo q and compared with the check symbols of the position sum. In Stage 3, the check symbols for the transfer digit vector are calculated using G and the final sum codeword is generated by adding the interim sum codewords and the transfer digits codewords modulo q. Error checking in this stage is performed by testing the validity of the final sum codeword using H.

While the encoding operation of a linear block code consists of multiplying a dataword vector with the generator matrix G, the error checking operation entails multiplying a codeword with the check matrix H. The hardware for encoding and error checking needs to implement a field multiplication of a vector with a matrix using the basic field addition and field multiplication between field elements. Since there are only $q \times q$ possibilities for each field operation on $GF(q)$, the field addition and field multiplication can be easily implemented with negligible hardware overhead. A straightforward approach consists of storing all $q \times q$ results in a multi-valued ROM-based look up table. Furthermore, such encoding and error checking hardware is common across communication, storage and arithmetic subsystems; thus the hardware overhead is amortized across all these subsystems.

The major operation of error checking in every stage consists of the multiplication of a vector with the matrix G or H: in stage 1, the codeword of the position sum is multiplied with H to check for its validity; in stage 2, the check bits of the interim sum and the radix vector are calculated by multiplying the corresponding dataword with G; in stage 3, the dataword of the transfer digits is multiplied with G to generate check bits and the codeword of the final sum is multiplied with H to conduct a validation check.

In terms of computation latency, multiplying an n digit vector with a matrix is the bottleneck. Multiplying an n digit vector with each column of a matrix consists of: n 2-operand multiplications followed by an n-operand addition. The n multiplications can be performed in parallel in one cycle. These n results are then added using a binary tree structure, where the leaves are the results of the n multiplications and the root denotes the final sum. The latency of this binary tree based addition is $O(\log n)$. The process of error

checking using a linear block code does not interfere in the critical path of the arithmetic operations. Therefore, although the checking result might be delayed by a few clock cycles, the checking operation in itself does not affect the overall system performance.

4. ERROR CHECKING CAPABILITY

It is difficult to justify and verify the error detection/correction capability of any fault-tolerance technique proposed for nanotechnologies, since convincing experimental data are not available at this current research stage. However, with well-structured information redundancy approaches such as linear block codes, it is possible to provide a formal proof, not only to validate, but also thus to precisely quantify its fault tolerance capability. Therefore, even despite the lack of knowledge regarding the implementation details of the underlying nanotechnologies, the fault tolerance level of the proposed scheme can be securely guaranteed.

In this section, we provide such a formal proof to show that the proposed technique can guarantee a predefined $(d-1)$-fault detection capability in the carry save addition operation. Since a carry save addition can be divided into three stages as described in the previous section, the proof is constructed by validating the error checking capability in each stage. We first outline some preliminaries and setups for the proof; subsequently, the validation of the three stages is shown in three separate subsections.

Assuming we apply a linear block code $\mathcal{C}$ with the minimum Hamming distance d between any two codewords; then

$$\forall c_1, c_2 \in \mathcal{C}, |c_1 - c_2| \geq d.$$

The information redundancy in $\mathcal{C}$ should provide a $(d-1)$-error checking capability. We will show that, in each of the three stages of the carry save addition, any error that changes fewer than d digits of the result can be detected by the proposed scheme.

Assume in the carry save addition, the operand digits to be within the range [0, α]; then the position sum digits are within the range [0, 2α]. Suppose the linear block code is constructed on $GF(q)$, where q is a prime number $\geq 2\alpha$. Let us denote the dataword, codeword and check symbols for a vector v as $D(v)$, $C(v)$ and $K(v)$, respectively. The encoding process using a generator matrix $G = [I|P]$ can be expressed by $K(v) = D(v) \times P$ and $C(v) = D(v) \times G$. Since G is constructed in a systematic form, each codeword has its dataword as a prefix. Therefore, $C(v) = [D(v)|K(v)]$.

4.1 Validation of Error Checking for the Position Sum Calculation

A position sum codeword $C(ps) = C(OP_1) + C(OP_2)$ is calculated by adding the codewords of the two operands. Since the arithmetic addition is identical to the field addition, the resultant vector remains a valid codeword, i.e., $C(ps) \in \mathcal{C}$. When an error occurs and changes e digits of $C(ps)$, where $e < d$, we claim that the faulty position sum codeword is invalid, i.e.,

$$C'(ps) \notin \mathcal{C}.$$

This is because $|C'(ps) - C(ps)| = e < d, C(ps) \in \mathcal{C}$. Since the minimum Hamming distance between any two codewords in $\mathcal{C}$ is d, if $C'(ps) \in \mathcal{C}$, we would have

$$|C'(ps) - C(ps)| \geq d > e,$$

a contradiction.

Therefore, $C'(ps)$ cannot be a codeword in $\mathcal{C}$. The check in this stage performed with H will justify $C'(ps) \notin \mathcal{C}$ with a nonzero syndrome vector.

4.2 Validation of Error Checking for the Position Sum Decomposition

The dataword part of the position sum, $D(ps)$, is split into two parts: the interim sum dataword $D(is)$ and the transfer digits dataword $D(td)$.

$D(td)$ is used to deliver the radix vector dataword $D(rv)$ and the relationship $D(ps) = D(is) + D(rv)$ holds.

$$\begin{aligned} K(is) + K(rv) &= D(is) \times P + D(rv) \times P \\ &= (D(is) + D(rv)) \times P \\ &= D(ps) \times P \\ &= K(ps) \end{aligned}$$

Error checking in this stage is performed by checking the equality relationship:

$$K(ps) = K(is) + K(rv). \tag{2}$$

Suppose when a fault occurs, the position sum dataword $D(ps)$ is split into the erroneous $D'(is)$ and $D'(td)$. When

$$D'(is) - D(is) = e_1, D'(td) - D(td) = e_2, e_1 + e_2 < d$$

holds, we want to show that

$$K(ps) \neq K'(is) + K'(rv) \tag{3}$$

thus implying that the check in equation 2 does detect the fault.

Since the check symbols $K'(is)$ and $K'(rv)$ are generated by $D'(is) \times P$ and $D'(rv) \times P$, we have

$$[D'(is)|K'(is)] \in \mathcal{C}$$

and

$$[D'(rv)|K'(rv)] \in \mathcal{C}.$$

$[D'(ps)|K'(ps)]$ is the linear combination of the above two codewords under field addition; therefore, $[D'(ps)|K'(ps)] \in \mathcal{C}$.

Since $|D'(td) - D(td)| = e_2$ and $D'(rv)$ is derived from $D'(td)$ digit by digit, $|D'(rv) - D(rv)| = e_2$.

Since

$$\begin{aligned} |D'(is) - D(is)| &= e_1 \\ |D'(rv) - D(rv)| &= e_2 \\ D'(is) + D'(rv) &= D'(ps) \\ D(is) + D(rv) &= D(ps) \end{aligned}$$

We have:

$$0 \leq |D'(ps) - D(ps)| \leq e_1 + e_2 \leq d.$$

Since the error actually occurs in the transfer digits, and $D'(rv)$ is derived by multiplying $D'(td)$ with the radix r, then

$$|D'(ps) - D(ps)| \neq 0$$

$$D'(rv) + D'(is) = D'(ps) \neq D(ps)$$

Suppose $K'(is) + K'(rv) = K'(ps)$; if we can show

$$|K'(ps) - K(ps)| > 0$$

then inequality 3 holds since

$$K'(ps) \neq K(ps)$$

In fact,

$$|[D'(ps)|K'(ps)] - [D(ps)|K(ps)]| \geq d.$$

since we have $[D'(ps)|K'(ps)] \in \mathcal{C}$ and $[D(ps)|K(ps)] \in \mathcal{C}$, from the above proof, we have:

$$0 < |D'(ps) - D(ps)| < d,$$

thus

$$|K'(ps) - K(ps)| > 0$$

and the check in equation 2 can detect the fault.

4.3 Validation of Error Checking for the Final Sum

In the third stage, the final sum vector is calculated by adding the codewords of the interim sum and the transfer digit vector. Since the error checking mechanism in this stage is identical to the one in the first stage, the proof of stage 1 can be applied to this stage directly.

5. CONCLUSIONS

The proposed scheme achieves fault tolerance in arithmetic operations via information redundancy. The same strategy has been widely used for fault tolerance in interconnection and memory subsystems. Within such a unified fault tolerance system, data are stored, transmitted and calculated using linear block codewords. The same error checking/correction unit can be used to provide fault tolerance for the different components and during distinct operations of the system.

By using a linear block code generated by a systematic generator matrix, the proposed scheme achieves concurrent error checking without introducing any delays in the critical path of the arithmetic operation. Error checking is performed on the check symbol part of a codeword and does not interfere with the calculation of the datawords. Therefore, the advantage of fast addition with constant delay provided by carry save arithmetic can be fully preserved.

The proposed approach provides flexibility to adjust the fault tolerance and redundancy levels in two ways. First, different levels of fault tolerance can be achieved by constructing linear block codes with a desired Hamming distance. Second, tradeoffs can be made between the amount of decode/encode hardware and error checking delay.

The proposed technique utilizes the multi-valued logic support in nanotechnology, yet does not depend on the specific underlying technology in the implementation of the multi-valued logic. As a rather general approach, it can be applied to any nanotechnology that supports multi-valued logic, possibly based on the negative differential resistance (NDR) characteristic.

6. REFERENCES

[1] M. S. Montemerlo, J. C. Love, G. J. Opitech, D. G. Gordon and J. C. Ellenbogen, *Technologies and Designs for Electronic Nanocomputers*, MITRE, July 1996.

[2] European Commission, *Technology Roadmap for Nanoelectronics*, 2001.

[3] K. Nikolic, A. Sadek and M. Forshaw, "Architectures for Reliable Computing with Unreliable Nanodevices", in *IEEE-NANO*, pp. 254–259, 2001.

[4] P. Beckett and A. Jennings, "Towards Nanocomputer Architecture", in *Asia-Pacific Computer System Architecture Conference*, pp. 141–150, 2002.

[5] J. von Neumann, "Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components", in C. Shannon and J. McCarthy, editors, *Automata Studies*, Princeton University Press, 1956.

[6] J. Han and P. Jonker, "A System Architecture Solution for Unreliable Nanoelectronic Devices", *IEEE Transactions on Nanotechnology*, vol. 1, n. 4, pp. 201–208, December 2002.

[7] J. R. Heath, P. J. Kuekes, G. S. Snider and S. Williams, "A Defect-Tolerant Computer Architecture: Opportunities for Nanotechnology", *Science*, vol. 280, pp. 1716–1721, June 1998.

[8] D. Mange, M. Sipper, A. Stauffer and G. Tempesti, "Toward Robust Integrated Circuits: The Embryonics Approach", *Proceedings of the IEEE*, vol. 88, n. 4, pp. 516–541, April 2000.

[9] W. J. Townsend, J. A. Abraham and E. E. Swartzlander, "Quadruple Time Redundancy Adders", in *DFT*, pp. 250–256, 2003.

[10] J. H. Patel and L. Y. Fung, "Concurrent Error Detection in ALUs by Recomputing with Shifted Operands", *IEEE Transactions on Computers*, vol. 31, pp. 589–592, December 1982.

[11] W. J. Townsend, J. A. Abraham and P. K. Lala, "On-Line Error Detecting Constant Delay Adder", in *IOLTS*, pp. 17–22, 2003.

[12] M. Nicolaidis, R. O. Duarte, S. Manich and J. Figueras, "Fault-Secure Parity Prediction Arithmetic Operators", *IEEE Design and Test of Computers*, vol. 14, n. 2, pp. 60–71, April-June 1997.

[13] T. Kim, W. Jao and S. Tjiang, "Arithmetic Optimization using Carry-Save-Adders", in *DAC*, pp. 433–438, June 1998.

[14] P. Mazumder, S. Kulkarni, M. Bhattacharya, J. P. Sun and G. I. Haddad, "Digital Circuit Applications of Resonant Tunneling Devices", *Proceedings of the IEEE*, vol. 86, n. 4, pp. 664–686, April 1998.

[15] J. Chen, M. A. Reed, A. M. Rawlett and J. M. Tour, "Observation of a Large On-Off Ratio and Negative Differential Resistance in an Electronic Molecular Switch", *Science*, vol. 286, pp. 1550–1552, 1999.

[16] A. F. Gonzalez and P. Mazumder, "Compact Signed-Digit Adder Using Multiple-Valued Logic", in *Conference on Advanced Research in VLSI*, pp. 96–113, 1997.

[17] S. Wei and H. C. Lin, "Multivalued SRAM Cell Using Resonant Tunneling Diodes", *IEEE Journal of Solid-State Circuits*, vol. 27, n. 2, pp. 212–216, February 1992.

[18] T. Kuo, H. C. Lin, R. C. Potter and D. Schupe, "Multiple-Valued Counter", *IEEE Transactions on Computers*, vol. 42, n. 1, pp. 106–109, January 1993.

[19] T. Waho, K. J. Chen and M. Yamamoto, "A Novel Multiple-Valued Logic Gate Using Resonant Tunneling Devices", *IEEE Electron Device Letters*, vol. 17, n. 5, pp. 223–225, May 1996.

[20] C. S. Wallace, "A Suggestion for a Fast Multiplier", *IEEE Transactions on Electronic Computers*, vol. 13, pp. 14–17, 1964.

[21] R. B. Blahut, *Algebraic Codes for Data Transmission*, Cambridge University Press, 2002.

[22] B. Parhami, *Computer Arithmetic Algorithms and Hardware Designs*, Oxford University Press, 2000.

[23] L. J. Micheel, A. H. Taddiken and A. C. Seabaugh, "Multiple-Valued Logic Computation Circuits using Micro- and Nanoelectronic Devices", in *International Symposium on Multiple-Valued Logic*, pp. 164–169, 1993.