TIME-SHARED TMR FOR FAULT-TOLERANT CORDIC PROCESSORS

Jae-Hyuck Kwak[§], *Vincenzo Piuri**, *and Earl E. Swartzlander, Jr.*[§]

[§] Department of Electrical and Computer Engineering, University of Texas at Austin, USA
* Department of Electronics and Information, Politecnico di Milano, Italy

ABSTRACT

This paper presents a low-cost approach to concurrent error correction in high-performance CORDIC processors by using time-shared triple modular redundancy. Operands are partitioned into three sets of disjoint digits and operations are performed three times on different hardware components to correct possible errors by majority voting. The approach has limited latency increase and throughput reduction. Pipelining can be used to maintain the same throughput as a conventional design.

1. INTRODUCTION

The CORDIC algorithm is an effective iterative technique for vector rotation and for evaluating more than a dozen of elementary functions [1]. Several dedicated VLSI architectures have been proposed in the literature to achieve high performance in real-time signal-processing applications. Many solutions are based on use of the redundant data representations [2, 3]. These techniques can be effectively applied only if the rotation directions can be estimated in advance, so that the high performance of the arithmetic paths is preserved. To predict the direction a few MSDs (most significant digits) can be inspected [4]; some architectures of this kind are in [3, 5, 6].

In many mission-critical applications (e.g., in aerospace and telecommunication), fault tolerance is mandatory. Whenever decisions, which are critical for the safety either of personnel or equipment, are taken on the basis of CORDIC processor outputs, these data must be correct. Similar needs occur when maintenance is difficult or impossible. On-line correction of all results can be accomplished by concurrent techniques. In the literature, many approaches are available for arithmetic processing arrays and other arithmetic structures, e.g., hardware redundancy, arithmetic data coding, and time redundancy [9, 10]. Usually, modular hardware redundancy has high performance, but also high circuit complexity. Arithmetic coding may have reasonable hardware redundancy and performance decrease, but only for quite limited fault coverage. Time redundancy may have low performance and complexity, but high fault coverage, especially for transient faults.

In this paper, we present an effective way for using Time-Shared Triple Modular Redundancy (TS-TMR) [11-13] to introduce fault tolerant capabilities to high-performance CORDIC processors. Time-shared TMR is an extension of the technique proposed in [14] for concurrent error detection in adders. In particular, we deal with the case of concurrent error correction. The main innovation of this paper consists of the use of this technique (which is typically used with purely arithmetic data paths) also for data paths containing conditional operations (as in the CORDIC algorithm). The proposed architecture is based on an unpipelined structure for redundant CORDIC as well as on the prediction scheme for rotation direction introduced in [4, 6]. A good trade-off between circuit complexity and performance is achieved by applying this technique at the system level. The error model encompasses all possible errors (both permanent and transient) in the output of a single module, implying that multiple faults at the gate level can be corrected if they affect only one module output. The error coverage is larger than that which is achieved with other techniques (like coding) at a similar level of circuit complexity. Throughput can be increased by pipelining.

Section 2 summarizes the CORDIC operation and the basic processor structure. Section 3 introduces the TS-TMR technique and its use for concurrent error correction. Cost and performance evaluation are given in Section 4.

2. BASIC CORDIC ARCHITECTURE

The unified CORDIC algorithm [1, 2] is defined by,

$$\begin{aligned} x_{i+1} &= x_i - m \cdot \sigma_i \cdot 2^{-S(m,i)} \cdot y_i \\ y_{i+1} &= y_i + \sigma_i \cdot 2^{-S(m,i)} \cdot x_i \\ z_{i+1} &= z_i - \sigma_i \alpha_{m,i} \end{aligned} \qquad (1)$$

where m denotes a coordinate system (+1: circular, 0: linear, or –1: hyperbolic), σ_i the rotation direction, and $S(m,i)$ the iteration sequence. $\alpha_{m,i}$ is the rotation angle defined as

$$\alpha_{m,i} = (1/\sqrt{m}) \cdot \tan^{-1}(\sqrt{m} \cdot 2^{-S(m,i)}) .$$

In rotation mode, the processor performs planar vector rotation, while in vectoring mode the magnitude and the angle of an initial vector are computed. σ_i is thus determined by

$$\sigma_i = \begin{cases} sign(z_i), & \text{for } \textit{rotation} \text{ mode } (z_n \to 0) \\ sign(-x_i \cdot y_i), & \text{for } \textit{vectoring} \text{ mode } (y_n \to 0) \end{cases} \qquad (2)$$

Estimation of the rotation direction in vectoring mode is usually based on the parameter w_i, defined as $w_i = 2^i \cdot y_i$ [2]. Therefore, the CORDIC iteration equations become

$$\begin{aligned} x_{i+1} &= x_i - \sigma_i \cdot 2^{-2i} \cdot w_i \\ w_{i+1} &= 2(w_i + \sigma_i \cdot x_i) \\ z_{i+1} &= z_i - \sigma_i \alpha_i \end{aligned} \qquad (3)$$

To predict σ_i, the estimated value of w_i, $\hat{w}_i$, is first produced by truncating w_i to t MSDs [4]; σ_i is then given by the sign of $\hat{w}_i$. For 16- and 32-bit operands, t is equal to 6 and 8, respectively [4]. This technique was efficiently applied in [6] to achieve a very high performance, by fully overlapping the computation along the arithmetic data paths and the rotation direction selection. Pipelining can be adopted to increase the throughput at different granularities with different cost/performance trade-offs [8]. The finer is the granularity, the higher the throughput, the circuit complexity and the latency.

3. CONCURRENT ERROR CORRECTION APPROACH

The basic concept of time redundancy [9, 10] is to perform the same computation several times in order to detect or correct

errors. Recomputation with the same functional modules and input operands can detect or correct transient failures only, not permanent ones. To deal with permanent failures, it is necessary either to change the functional modules or the operands.

Time-shared TMR has been shown an effective technique for concurrent error correction in arithmetic systems [11-15], at reasonable circuit complexity. Basic arithmetic operations (addition and multiplication) are divided in three parts by splitting the operands: each part of the operands is treated by the corresponding partial operation. Fault tolerance is achieved by performing the partial operations by using three circuits in parallel. Majority voting determines the correct output of each partial operation in the presence of any kind of fault confined to any of the functional modules. A self-checking voter is used.

Time-shared TMR is more efficient from the points of view of circuit complexity, latency, and throughput if voting is performed at the highest level of the system. This is possible whenever carries do not need to be propagated immediately to the most significant parts, i.e., when the overall computation can be viewed as a cascade of arithmetic operations and the carry propagation delayed till the last stage of the nominal operations. This is the case of the inner product and the convolution [15], the FFT algorithm [13], and the data path in the CORDIC processor. In these cases, we can avoid voting and storing the intermediate carries within the individual step of the envisioned DSP algorithm. This results in saving circuit complexity and latency.

3.1. Carry propagation in the arithmetic data paths and rotation prediction for Time-Shared TMR

The CORDIC architecture without fault tolerance abilities essentially consists of three arithmetic data paths and the rotation direction generators for each CORDIC stage. Each arithmetic data path is composed by adders/subtractors and hardwired shifters. Carry propagation within the arithmetic data paths is needed both to select the rotation direction at each CORDIC stage and to generate the final outputs. The final carry propagation produces the CORDIC results in the conventional binary representation.

Carry propagation within a CORDIC stage is necessary in order to produce the information to be used for rotation selection in the subsequent CORDIC stage. This would make it impossible to use TS-TMR effectively since each CORDIC stage should wait for completion of the iterations and, consequently, for generation of the whole output of the CORDIC stage. Latency will triplicate with respect to the non fault tolerant case; circuit complexity will increase massively due to the additional registers.

However, in [4] it was proven that only a limited number t of digits is actually needed to predict the rotation direction accurately. Let's consider the first CORDIC stage of the iteration in which the most-significant parts of the operands are processed. Let's assume that the number of digits, n, that is processed by this iteration is not smaller than the value t [4] required for rotation direction prediction on the total number, b, of the operand bits. (If this condition does not hold, we simply consider the smallest of the higher numbers of operand bits that satisfies this condition.) By applying the TS-TMR technique as presented above for adders, n should be equal to $\lceil b/3 \rceil$. The most-significant t bits of the first CORDIC stage are the same that are used both in the prediction schemes described in [4, 6] and in the iteration performed on most-significant n-bit operand parts. The rotation direction of the first CORDIC stage will be the same.

Let's now consider the subsequent CORDIC stages. The higher the index of the CORDIC stage, the more the carries need to be propagated from the middle-significant part of the computation to guarantee the same accuracy of the prediction over the t most-significant digits. For b-bit operands (and b CORDIC stages), we need $\lceil \log_2 b \rceil$ guard bits to ensure the same accuracy of the t most-significant bits. The most significant n-bit part of the operands to be processed during the iteration consists of at least of $t + \lceil \log_2 b \rceil$ bits. In the case of $b=32$, n is 13 bits.

3.2. Partitioning in the arithmetic data paths for Time-Shared TMR and execution order

By partitioning the CORDIC operands in three *n-bit* parts with $n = t + \lceil \log_2 b \rceil$, the rotations computed on the most-significant operand part in all iterations will be identical to the ones generated in [4, 6] by processing all the operand bits. This makes the computation on the most-significant operand parts independent from the computation of the other parts. Carry propagation is confined to the final adding stage.

Conversely, computation of each CORDIC stage on the middle- and least- significant parts of the operands still depends on the rotation directions decided during the CORDIC iteration on the most-significant parts of the operands. The rotation predicted in each CORDIC stage during such any iteration holds also for the other iterations on the middle- and the least-significant parts. The rotation directions must be computed during the iteration on the most significant parts of the operands, stored within each stage, and then used during the subsequent iterations on the middle- and the least- significant parts. The need of predicting first the rotation directions on the most-significant parts of the operands induces a partial order in the execution of the iterations. The most significant iteration must precede the other two. No execution order is implied by rotation direction prediction between the iteration on the middle- and least-significant parts.

This execution order differs from the one usually adopted, e.g., see [11, 13, 15]. In these latter cases, the natural execution order is due to the ease of carry management and propagation in the final addition. Starting from the least-significant digits allows for partially overlapping the carry propagation of digits generated during one iteration with the computation performed in the subsequent iteration on most significant digits, thus resulting in low latency. In the above execution order, the order that naturally follows the carry propagation cannot be adopted since rotation direction prediction must be performed before any computation.

Since the prediction does not force any execution order between the iterations on the middle- and the least- significant parts of the operands, we can still adopt the natural order suggested by the carry propagation for these two iterations so as to minimize the overall latency. The bits that must be processed by these two iterations are $b - n = b - t - \lceil \log_2 b \rceil$. Usually, this number of bits is higher than n. To maximize the use of hardware devices, one of the iterations will process n bits, while the other

will deal with $b-2t-2\lceil \log_2 b \rceil$ bits. In the case of *b=32*, two iterations will process 13 bits and the third one will work on 6 bits only.

In summary, the first iteration will process the *n* most-significant bits of the operands – three times in parallel – to generate the rotation directions and the most significant contribution to the final CORDIC result. The second iteration will process the *n* least-significant bits to generate the least significant part of the final CORDIC result. While carries are propagated in the positions exceeding the *n* least significant ones, the third iteration will process the remaining $b-2t-2\lceil \log_2 b \rceil$ middle-significant bits. The bits in these positions are added to the carries produced by the second iteration to produce the middle part of the final CORDIC result. Carries exceeding the $b-t-\lceil \log_2 b \rceil$ position are added to the contribution computed by the first iteration to generate the most significant part of the result. Such an addition can start as soon as the output bits corresponding to the $b-2t-2\lceil \log_2 b \rceil$ input bits are generated since the other bits are not relevant. The three copies of the final CORDIC result are voted to correct possible errors.

3.3. The rotation direction generators

The irregularity of the structure of the rotation direction generators [4, 6] and the presence of non-arithmetic functions in them make replication [9] more practical for these circuits. Complexity of such circuits is very low with respect to the adders in the three arithmetic data paths. An error in the computation of any rotation direction appears (if not masked) as an erroneous final result produced by the circuit computing the corresponding iteration. Under the single-module fault assumption, at most one final output will be affected by a fault that occurs in one of the rotation direction generators. Therefore, even if rotation directions are not directly checked and corrected within each stage, errors due to a fault in one of the direction generators are still detected and corrected at the final outputs. To limit the circuit complexity without impairing the correction abilities, we introduce one rotation direction generator in each stage of every iteration. The three copies of the rotation direction generated in each stage are not voted but used independently in the corresponding iteration since possible errors will be indirectly detected and corrected by voting the replicas of the final results.

3.4. The fault tolerant architecture

The resulting architecture with concurrent error correction ability is shown in Figure 1. Essentially, it is obtained by tripling the non fault tolerant architecture. More precisely, the fault-tolerant architecture consists of three triplets of *n*-bit data paths: each triplet realizes the computation of the corresponding *b*-bit data path in the nominal architecture. The *n*-bit data paths of a triplet operate in parallel on the corresponding inputs: each of them processes in parallel the same portion (most-, middle-, or least-significant) of the operands during each iteration. The internal structure of each data path is identical to the nominal one (except that the data word size is reduced). The proposed approach can be adopted with any type of adders/subtractors in each stage (i.e., ripple-carry, carry-look-ahead, conditional-sum, carry-save units). Registers in each stage hold the portions of the intermediate results that have been produced by the first iteration in the *x* and *y* data paths and that are needed in the subsequent stage in the other iterations of the *y* and *x* data paths.

Each stage contains three copies of the rotation direction generator. One is used for each iteration, the internal structure of these generators is identical to that which is used in [4, 6]. Rotations independently computed during the first iteration are stored (without being voted) in registers to be used during the subsequent two iterations. The contribution to the final results generated by the stages during the first iteration are stored in registers until they can be added to the least significant portions. Similarly, the contribution of the second iteration is stored in other registers until the middle-significant part of the results become available in the third iteration. The merging and voting module at the outputs of the stages provides the final carry propagation and the appropriate merging of the intermediate partial results produced by the iterations. Voting corrects any single error in the three values of each output.

To minimize the latency, conditional-sum based structures can be adopted for the arithmetic units [6]. Computation in data paths and generation of the rotation direction can thus be parallelized. To enhance the throughput, pipelining can be implemented by placing pipeline registers between each pair or stages, as shown in [8]. Registers are implicitly protected by the use of TS-TMR. The error correction ability is identical in all these architectures.

4. PERFORMANCE AND CONCLUSION

The design of a CORDIC architecture with concurrent error correction abilities has been presented by using Time-Shared Triple Modular Redundancy. This technique is typical of purely arithmetic data paths: we have shown how to apply it successfully also to data paths with conditioned operations. Rotation prediction schemes have been used to predict the rotation direction while processing only a portion of each operand, thus allowing for the application of TS-TMR.

The circuit complexity increase with respect to a conventional (i.e., non-fault-tolerant) design is about 100%. The latency increase is about 40% for ripple-carry adders, but it is well over 200% for carry-look-ahead, conditional-sum and carry-save units. The throughput reduction is about 30% for ripple-carry adders and about 70% for the others. The TS-TMR design has less circuit complexity than other modular redundancy techniques. It is also more effective than other time redundancy approaches when the latency increase is considered since the TS-TMR operands are shorter than the nominal ones.

5. REFERENCES

[1] J. S. Walther, "A Unified Algorithm for Elementary Functions," *Proc. of Spring Joint Computer Conference*, pp. 379-385, 1971.

[2] M. D. Ercegovac and T. Lang, "Redundant and On-line CORDIC: Application to Matrix Triangularization and SVD," *IEEE Trans. on Comput.*, vol. 39, pp. 725-740, 1990.

[3] N. Takagi, T. Asada, and S. Yajima, "Redundant CORDIC Methods with a Constant Scale Factor for Sine and Cosine Computation," *IEEE Trans. on Comput.*, vol. 40, pp. 989-995, 1991.

[4] J.-A. Lee and T. Lang, "Constant-Factor Redundant CORDIC for Angle Calculation and Rotation," *IEEE Trans. on Comput.*, vol. 41, pp. 1016-1025, 1992.

[5] D. Timmermann, H. Hahn, and B.J. Hosticka, "Low Latency Time CORDIC Algorithms," *IEEE Trans. on Comput.*, vol. 41, pp. 1010-1014, 1992.

[6] J.-H. Choi, J.-H. Kwak, and E. E. Swartzlander, Jr., "High-Speed CORDIC Architecture Based on Redundant Sum Formation and Overlapped Sigma-Selection," *IEEE Proc. of the International Conference on Computer Design*, 1999.

[7] S. Wang, V. Piuri, and E. E. Swartzlander, Jr., "Hybrid CORDIC Algorithm," *IEEE Trans. on Comput.*, vol. 46, pp. 1202-1207, 1997.

[8] S. Wang and V. Piuri, "A Unified View of CORDIC Processor Design," in *Application Specific Processing*, Boston: Kluwer Publishing, pp. 121-160, 1997.

[9] D.P. Siewiorek and R.S. Swarz, *Reliable computer systems: design and evaluation*, Burlington, MA: Digital Press, 1992.

[10] T.R.N. Rao, *Error Coding for Arithmetic Processors*, New York: Academic Press, 1974.

[11] Y.-M. Hsu and E.E. Swartzlander, Jr., "VLSI concurrent error correcting adders and multipliers," *Proc. 1993 IEEE Int'l Workshop on Defect and Fault Tolerance in VLSI Systems*, pp. 287-294.

[12] Y.-M. Hsu, V. Piuri, and E.E. Swartzlander, Jr., "Efficient time redundancy for error correcting inner-product units and convolvers," *Proc. 1995 IEEE Int'l Workshop on Defect and Fault Tolerance in VLSI Systems*, pp. 287-294.

[13] V. Piuri and E.E. Swartzlander, Jr., "Time-Shared Modular Redundancy for Fault-Tolerant FFT Processors," *Proc. 1999 IEEE Int'l Symposium on Defect and Fault Tolerance in VLSI Systems*, 1999.

[14] B.W. Johnson, J.H. Aylor, and H.H. Hana, "Efficient use of time and hardware redundancy for concurrent error detection in a 32-bit VLSI adder," *IEEE J. Solid-State Circuits*, Vol. 23, pp. 208-215, 1988.

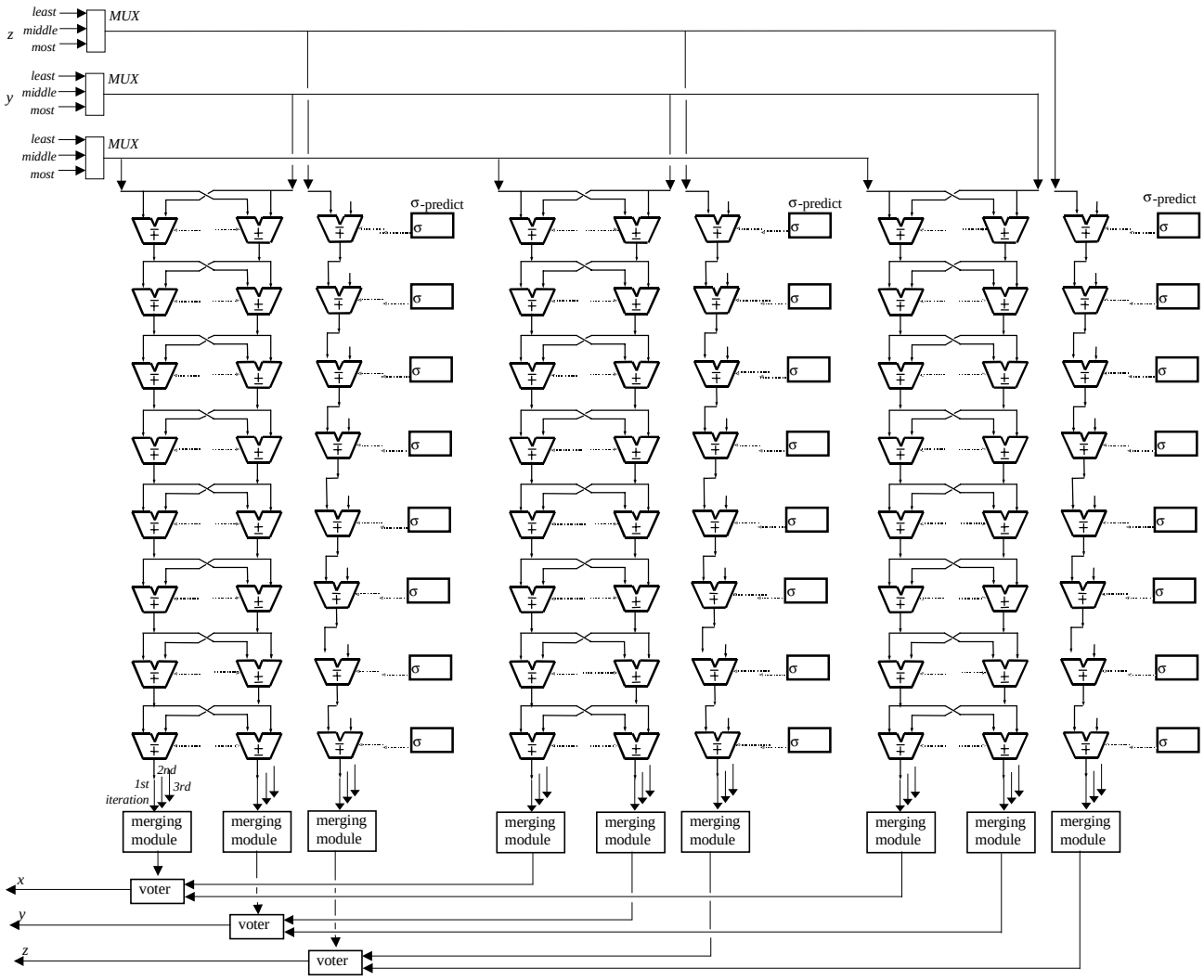

Figure 1. The CORDIC structure with concurrent error correction ability.