Algorithm-Based Fault Tolerance for Floating-Point Operations in Massively Parallel Systems*

Jennifer Rexford[◇] and *Niraj K. Jha*[†]

◇ Dept. of EECS, Univ. of Michigan, Ann Arbor, MI 48109
† Dept. of EE, Princeton Univ., Princeton, NJ 08544

Abstract

This paper considers the applicability of algorithm-based fault tolerance (ABFT) to massively parallel scientific computation. Existing ABFT schemes can provide effective fault tolerance at a low cost for computation on matrices of moderate size; however, the methods do not scale well to floating-point operations on large systems. This paper proposes the use of *a partitioned linear encoding* scheme to provide scalability. Matrix algorithms employing this scheme are presented and compared to current ABFT schemes. It is shown that the partitioned scheme provides scalable linear codes with improved numerical properties with only a small increase in hardware and time overhead.

*Acknowledgment: This work was supported in part by Office of Naval Research under Contract no. N00014-91-J-1199 and in part by Air Force Office of Scientific Research under Contract no. AFOSR-90-0144.

1 Introduction

Much energy has been directed in the last few years at developing various types of parallel processing systems. As technology improves, systems with thousands, even millions, of processing elements become feasible. Although these novel architectures put a great deal of power at the users' disposal, the possibility of transient hardware errors in long, complex computations can be paralyzing. As systems include more processors and tighter geometries, fault detection and fault tolerance assume greater importance. Since transient errors are nearly impossible to duplicate and detect, a truly reliable parallel system must respond to such errors at run-time.

Researchers have proposed a variety of methods for dealing with transient errors. In particular, a method called algorithm-based fault tolerance (ABFT) [1, 2] merits special attention since it can provide fault tolerance at a low cost, in terms of running time and hardware overhead. However, in general, ABFT can significantly complicate algorithm development. Fortunately, a wide variety of scientific and engineering applications depend on just a handful of related matrix operations. As a result, ABFT systems have been designed for several matrix-based computations, used frequently in signal-processing applications, such as matrix multiplication, matrix inversion, LU decomposition, and FIR filtering [1]-[5]. In addition, an algorithm-based approach has proved useful in developing fault-tolerant implementations of the FFT [4, 6, 7, 8], sorting [9], and median filtering [5]. Further research has resulted in methods for the analysis and design of ABFT systems [10]-[22].

Work has also been done in developing new codes for ABFT, such as generalized checksum codes [14] and the sum-of-squares code [4]. Huang and Abraham [1] suggest the checksum coding technique for the detection and correction of errors in matrix manipulations. In theory, this provides a low-cost method for achieving fault tolerance; however, any ABFT method must deal effectively with the numerical limitations of finite precision arithmetic. Thus, the method must permit a certain amount of error to account for numerical inaccuracy. However, as n (the number of elements whose checksum is required) increases, so does the estimated numerical error in the checksum. Thus, a transient fault in a single processing element becomes more difficult to detect. Likewise, a detected fault becomes more difficult to locate. Nair and Abraham [14] address some of these numerical problems, particularly overflow, by introducing *general linear codes.* Although these codes improve the numerical properties of ABFT schemes, the techniques are still extremely sensitive to roundoff errors, particularly in computations involving large matrices. This paper considers possible ways to scale linear code ABFT error detection techniques to floating-point matrix operations in massively parallel systems.

2 Linear Codes for ABFT

In this section we present some preliminary results on linear codes employed in ABFT systems.

2.1 Weighted Checksum Encoding

Many ABFT techniques utilize a general weighted checksum encoding for matrices [2]. In this representation, an $n \times n$ matrix is modified by adding to it additional rows and columns to store the row/column sums and weighted row/column sums. If $e^T = [1\,1 \ldots 1]$ and $e_w^T = [2^0\,2^1 \ldots 2^{n-1}]$, the $(n+2) \times (n+2)$ full weighted checksum matrix for the $n \times n$ matrix A, in block form, becomes

$$\begin{bmatrix} A & Ae & Ae_w \\ e^T A & e^T Ae & e^T Ae_w \\ e_w^T A & e_w^T Ae & e_w^T Ae_w \end{bmatrix}$$

as described in [2]. A column or row weighted checksum matrix would include only the column checksums or only the row checksums, respectively. The unweighted sums, of the form Ae or $e^T A$, are denoted as WCS1; the exponentially weighted sums, of the form Ae_w or $e_w^T A$, are denoted as WCS2.

A single fault, affecting only one matrix element, can be detected and located, and the resulting error corrected after a matrix operation using this scheme. For the location and correction of a single error, Jou and Abraham [2] introduce the variables

$S1$ and $S2$ such that

$$S1 = \sum_{j=1}^{n} a_j - WCS1$$

$$S2 = \sum_{j=1}^{n} 2^{j-1} a_j - WCS2$$

where $a = (a_1, a_2, \cdots, a_n)$ represents either a row or a column vector. $S1$ represents the *error* in the unweighted checksum; $S2$ denotes the *error* in the weighted checksum. If the two *errors* $S1$ and $S2$ are non-zero, $\frac{S2}{S1} = 2^{j-1}$ implies that a_j is the incorrect element. The faulty value may then be corrected by making $a_j = a_j - S1$. Using this scheme, Jou and Abraham [2] provide fault-tolerant algorithms for a number of signal processing and matrix operations.

2.2 Critique and Analysis

The work in [2] provides a strong theoretical background for the field of algorithm-based fault detection and tolerance; however, many important issues remain to be addressed. In particular, the numerical properties of ABFT schemes merit more rigorous analysis [15]. The algorithms must tolerate certain inaccuracies due to roundoff errors in floating-point arithmetic. The work in [2] assumes a fixed-point number system. Although fixed-point arithmetic may be necessary and practical for signal processing and other real-time applications (because of speed requirements), this does not provide general ABFT for scientific computation.

Extending the weighted checksum technique to floating-point arithmetic introduces a number of complications. First of all, the algorithms have to tolerate a certain error Δ (deviation from zero) in the values $S1$ and $S2$. The parameter Δ constitutes the largest roundoff error the scheme can encounter without signaling a fault [15]. If Δ is set too low, typical numerical errors could be misinterpreted as faults (*false alarms*), incurring the cost of fault location/correction when it is not truly needed [15]; however, setting Δ too high can significantly detract from fault coverage. Determining the appropriate tolerance level depends heavily on the nature of the data and the computation, as well as the size of the matrix involved. This makes it difficult to scale the work in [2] to massively parallel systems that utilize floating-point arithmetic.

Furthermore, the design of an ABFT scheme must include a study of the possibility of overflow. In particular, the weighted checksums Ae_w or $e_w^T A$ may be extremely large (especially for a large, dense matrix). Also, for large n, the size of the larger weights (2^{j-1} for j close to n) can obliterate the effect of the smaller weights (j close to 1), making the detection/correction of many errors virtually impossible.

Luk [16] proposes changing the weights in order to avoid the above problem. He suggests the weight $w_j = j$, instead of $w_j = 2^{j-1}$, in the weighted vector $e_w^T = [w_1\, w_2 \ldots w_n]$. Although these new weights do not grow large as quickly as 2^{j-1}, they still can become prohibitively large for n in the thousands or millions. Changing the structure of the weights to $w_j = j$ also introduces additional complications. The exponentially weighted checksums (WCS2) in [2] are easily computed in a floating-point system — since multiplication by a power of two can be performed simply by changing exponents, there is no implicit loss of precision. The linear weights $w_j = j$ require more time to multiply and can introduce numerical error. Also, the smaller spacing of the weights makes fault *location* more difficult. The division $\frac{S2}{S1} = j$ implies that the faulty element is a_j. However, an error of ± 0.5, due to the accumulation of roundoff error and error in checksum division, can lead to erroneous fault location.

2.3 General Linear Codes

In order to address the numerical limitations of ABFT codes, Nair and Abraham [14] introduced the notion of *general linear codes*. In many applications the best encoder vector depends heavily on the data. If all of the elements in a matrix are positive, for example, the checksum encoder vector $e^T = [1\,1 \ldots 1]$ and the weighted checksum encoder vector $e_w^T = [2^0\, 2^1 \ldots 2^{n-1}]$ could suffer from significant problems with overflow. A different vector, called the periodic encoder vector $p^T = [1\ -1\,1\ -1 \ldots]$, could virtually eliminate this problem [14]. Thus, a broad class of general linear codes allows encoder vectors to be tailored to match the data, if the algorithm designer has some prior knowledge about the nature of the data.

Nair and Abraham also discuss the importance of code *reflectivity*. Code reflectivity is the ratio of the change in the coded value to the change in the data element, which caused the change in the coded value [14]. High reflectivity is crucial, particularly in error correction. Thus, the degree of reflectivity constitutes an important factor in choosing the appropriate linear encoder vector for an ABFT scheme.

In addressing the problem of numerical error, the work in [14] asserts that the upper bound on the roundoff error in encoding is dependent on the Euclidean norm of the encoder vector. The roundoff error, E, from encoding satisfies

$$E \leq um\|g^T\|_2\|x\|_2$$

where u and m are machine-dependent parameters for the unit roundoff and the mantissa, respectively, $g^T = (g_1, g_2, \cdots, g_n)$ is the encoder vector, and $x = (x_1, x_2, \cdots, x_n)$ is the vector to be encoded [23].

Consider the general linear encoder vector g^T with

$$\|g^T\|_2 = (|g_1|^2 + |g_2|^2 + \ldots + |g_n|^2)^{\frac{1}{2}}$$

for a vector of length n. For example, the weighted checksum encoder vector e_w^T has a large Euclidean norm

$$\|e_w^T\|_2 = \left(\sum_{j=0}^{n-1}(2^j)^2\right)^{\frac{1}{2}} = \left(\frac{4^n - 1}{3}\right)^{\frac{1}{2}} \simeq 0.6 \cdot 2^n$$

which explains the code's poor numerical properties, as discussed above, especially for large n. This is also true, although to a lesser extent, for the unweighted encoder vector e^T, which has $\|e^T\|_2 = \sqrt{n}$. A direct way to reduce the encoding error is to minimize $\|g^T\|_2$. Thus, a good linear code effectively balances the conflicting demands for high reflectivity and low Euclidean norm [14]. However, one way to alleviate this problem is to decrease the length of the encoder vector without changing its general structure. This suggests incorporating general linear codes into a partitioning scheme, thereby enhancing the numerical properties of ABFT encoding methods.

3 Partitioned Encoding for Matrices

In this section we discuss our encoding scheme.

3.1 Encoding Method

A linear code partitioning scheme can be used to scale ABFT techniques to operate effectively on massively parallel systems. Suppose matrix algorithms employing a particular coding scheme, using k encoder vectors, can capably handle operations on matrices of size $m \times m$ without being stymied by numerical error. Then, partition any $n \times n$ matrix into $(\frac{n}{m})^2$ $m \times m$ matrices, where we assume n is a multiple of m. Convert each of these submatrices into a fully coded matrix of size $(m+k) \times (m+k)$. So, this mechanism requires a total of $[\frac{n}{m}(m+k)]^2$ elements, whereas the matrices in [2] require $(n+k)^2$ elements.

In the past, partitioning of a matrix has been employed to allow a small system to handle a larger matrix, one submatrix at a time [1]; however, in our method the system operates on the entire matrix — the *encoding* is partitioned, in order to improve the numerical properties of the scheme. For example, suppose k $1 \times m$ encoder vectors g_i^T, $i = 1, \ldots, k$, are used to encode an $n \times n$ matrix A with $n = pm$, then the matrix takes the form

$$\begin{bmatrix} A_{11}^c & A_{12}^c & \cdots & A_{1p}^c \\ A_{21}^c & A_{22}^c & \cdots & A_{2p}^c \\ \vdots & \vdots & \ddots & \vdots \\ A_{p1}^c & A_{p2}^c & \cdots & A_{pp}^c \end{bmatrix}$$

where each A_{ij}^c is an $(m+k) \times (m+k)$ encoding of the $m \times m$ submatrix A_{ij} of A, such that

$$A_{ij}^c = \begin{bmatrix} A_{ij} & A_{ij}g_1 & \cdots & A_{ij}g_k \\ g_1^T A_{ij} & g_1^T A_{ij} g_1 & \cdots & g_1^T A_{ij} g_k \\ \vdots & \vdots & \ddots & \vdots \\ g_k^T A_{ij} & g_k^T A_{ij} g_1 & \cdots & g_k^T A_{ij} g_k \end{bmatrix}$$

In particular, consider the $n \times n$ matrix B with $n = 2m$ and $k = 1$. The matrix B has four submatrices B_{11}, B_{12}, B_{21}, and B_{22}, each fully encoded with the encoder vector g^T. The partitioned encoded matrix for B, in block form, is

$$\begin{bmatrix} \begin{bmatrix} B_{11} & B_{11}g \\ g^T B_{11} & g^T B_{11}g \\ B_{21} & B_{21}g \\ g^T B_{21} & g^T B_{21}g \end{bmatrix} & \begin{bmatrix} B_{12} & B_{12}g \\ g^T B_{12} & g^T B_{12}g \\ B_{22} & B_{22}g \\ g^T B_{22} & g^T B_{22}g \end{bmatrix} \end{bmatrix}$$

A partitioned encoded matrix with k encoder vectors has H times more elements than a regular encoded matrix, where

$$H = \frac{(\frac{n}{m}(m+k))^2 - (n+k)^2}{(n+k)^2} = \left(\frac{1+k/m}{1+k/pm}\right)^2 - 1$$

Therefore, if $p \gg 1$, $H \simeq (1+k/m)^2 - 1 = k^2/m^2 + 2k/m$, or since k/m is usually quite small

$$H \simeq \frac{2k}{m}$$

independent of n. If p is not much larger than 1, then the value of H would be even smaller. Thus, for a reasonable base matrix of size $m \times m$, the additional hardware overhead for the *partitioned* encoded matrix is small. For example, suppose a weighted checksum scheme could capably handle matrices of size 200×200. With $m = 200$ and $k = 2$ (for the two encoder vectors e^T and e_w^T), the partitioned checksum matrix has only 2% more elements than the traditional full checksum matrix. Thus, the time required to operate on the partitioned matrix is not significantly larger than the time needed in the traditional encoding scheme.

3.2 Implementation of Partitioned Encoding

The partitioned encoding scheme requires some changes in hardware support for ABFT. For the method to be worthwhile, the hardware overhead must be small, and scale well to large systems.

3.2.1 Linear Array

Consider an n-processor linear array which provides no algorithm-based fault detection. If a traditional (non-partitioned) checksum encoding is provided, the array must include one additional processor for the checksum. In addition, the system must include some means of calculating and comparing the single checksum. Although an n-input adder could provide this support, a tree of adders is more reasonable for large n. A tree of two-input adders would have $\lceil log_2 n \rceil$ stages and employ $n-1$ adders. Thus, the traditional checksum scheme would require $n+1$ processors, $n-1$ adders, and one totally self-checking comparator; the checksum computation would require $\lceil log_2 n \rceil$ adder delays.

A linear array utilizing the *partitioned* checksum encoding scheme (with partitions of size m) would require $n + \frac{n}{m}$ processing elements — $1/m$ times more than for an array with no fault detection. In order to compute the partitioned checksums in parallel for each of the n/m partitions, the system should provide $\frac{n}{m}$ m-input adder trees, one for each checksum. Each such tree requires $\lceil log_2 m \rceil$ stages and employs $m-1$ adders and one comparator; thus, the total number of adders is $\frac{n}{m}(m-1) = n - \frac{n}{m}$ — actually *less* than in the traditional encoding scheme. In addition, the checksum computation requires *less* time, since the partitioned scheme has less adder delays.

In summary, for large n, implementing the partitioned scheme in a linear array requires roughly $1/m$ times more processing elements and comparators than with the traditional encoding method. Both of these values scale well for large n, since m is constant. As a benefit, the partitioned encoding also requires $1/m$ times less adders and $(\lceil log_2 n \rceil - \lceil log_2 m \rceil)$ less adder delays. In particular, the smaller time required for checksum computation makes the partitioned scheme particularly attractive for very large n; the time required to compute checksums in the partitioned scheme scales well to massively parallel systems.

In addition, the partitioning scheme allows for multiple fault detection. In a linear array, the checksum (weighted checksum) technique allows for the detection (location and error correction) of a single fault [2]. However, in the partitioned linear array up to n/m faults can be detected simultaneously, and the resulting errors corrected if partitioned weighted checksum encoding is used, if each partition has no more than one fault. As vector length increases, the possibility of multiple transient faults grows as well. The partitioned codes allow a linear array to tolerate a fault in each subvector of length m, where m is independent of n.

3.2.2 Mesh Array

We next consider an ABFT implementation on a two-dimensional mesh of processors. A regular checksum encoding requires an $(n+1) \times (n+1)$ grid of processors for an $n \times n$ matrix. The partitioned encoding scheme for checksum encoding ($k = 1$) requires a $\frac{n}{m}(m+1) \times \frac{n}{m}(m+1)$ array, which contains roughly $2/m$ times more hardware than the non-partitioned encoding method. The checksums for each row/column can be computed by processing elements in other rows/columns [1]; thus, no additional hardware is required to compute the checksums. This also adds flexibility to the partitioned encoding scheme — since computing the checksums for the submatrices does not require special hardware, the value of m need not be embedded in the system. The parameter m can change in order to tailor the algorithm to the data, if necessary, to provide additional numerical stability.

As is the case with the partitioned linear array implementation, the partitioned mesh array approach can provide improved multiple fault coverage. The nonpartitioned technique in [1] allows the system to locate a single fault and correct the resulting error, or detect as many as three faults. In the partitioning scheme, a single fault can be located and the resulting error corrected, or three faults can be detected, in each $m \times m$ submatrix. So, the partitioned mesh array can locate up to $(n/m)^2$ faults and correct errors due to them, or detect as many as $3(n/m)^2$ faults, if each $m \times m$ submatrix contains no more than one or three faults, respectively.

3.3 Computation with Partitioned Encoded Matrices

A wide variety of important scientific and signal processing applications can be expressed in terms of matrix-matrix or matrix-vector addition or multiplication. For the scheme developed above to be worthwhile, there must be partitioned matrix addition and multiplication algorithms that do not require too much time overhead, as compared with the traditional encoding methods.

Huang and Abraham [1] proved that the addition of two (non-partitioned) checksum matrices derived from unencoded matrices A and B produces another checksum matrix, for matrix C, where $C = A + B$. Likewise, they showed that the product of a column checksum matrix derived from A and a row checksum matrix derived from B produces a full checksum matrix for C, where $C = A * B$. These results can be generalized for any linear code.

Since matrices can be added or multiplied in blocks, as long as the block sizes are consistent, the sum of two partitioned coded matrices can be expressed as the sum of the corresponding coded submatrices; likewise, the product of two partitioned coded matrices can be expressed in terms of the multiplication and addition of the coded submatrices. Thus, matrix addition and multiplication preserve the partitioned checksum property, as do the scalar product and transpose of a matrix. In matrix multiplication, the $m \times m$ submatrices of A should be column-encoded, whereas the submatrices of B should be row-encoded. For example, when $n = 2m$ and $k = 1$, using encoder vector g^T, we have:

$$\begin{bmatrix} A_{11} & A_{12} \\ g^T A_{11} & g^T A_{12} \\ A_{21} & A_{22} \\ g^T A_{21} & g^T A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{11}g & B_{12} & B_{12}g \\ B_{21} & B_{21}g & B_{22} & B_{22}g \end{bmatrix}$$

$$= \begin{bmatrix} C_{11} & C_{11}g & C_{12} & C_{12}g \\ g^T C_{11} & g^T C_{11}g & g^T C_{12} & g^T C_{12}g \\ C_{21} & C_{21}g & C_{22} & C_{22}g \\ g^T C_{21} & g^T C_{21}g & g^T C_{22} & g^T C_{22}g \end{bmatrix}$$

where $C_{11} = A_{11}B_{11} + A_{12}B_{21}$, $C_{12} = A_{11}B_{12} + A_{12}B_{22}$, $C_{21} = A_{21}B_{11} + A_{22}B_{21}$, and $C_{22} = A_{21}B_{12} + A_{22}B_{22}$.

The algorithms for matrix addition and multiplication do not have to change to incorporate the partitioned encoding scheme. The only computational difference is the size of the matrices involved. As discussed before, a full partitioned encoded representation for the $n \times n$ matrix A requires only $2k/m$ times more elements than the traditional encoding. Thus, the computational stage of the ABFT algorithm does not require a significant increase in time or complexity to accommodate the partitioned encoding scheme.

Likewise, the partitioned encoding representation is preserved in LU decomposition. When a matrix A is LU decomposable, the full partitioned encoded checksum matrix of A can be decomposed into a partitioned column-encoded lower triangular matrix L_c and a partitioned row-encoded upper triangular matrix U_r. This is true, as in [1], because matrix multiplication preserves the encoding. The product of L_c and U_r yields the fully partitioned encoded matrix of A. Thus, the basic operations of matrix addition, multiplication, scalar product, transpose, and LU decomposition preserve the partitioned encoding.

3.4 Numerical Properties

The partitioned checksum encoding has improved numerical properties. The weighted checksum encoder vector e_w^T has a large Euclidean norm $\|e_w^T\|_2 \simeq 0.6 \cdot 2^n$ for matrices of size $n \times n$. In the partitioned scheme with submatrices of size $m \times m$, where m is a constant,

$$\|e_w^T\|_2 \simeq 0.6 \cdot 2^m \ll 0.6 \cdot 2^n$$

for $m \ll n$. This puts a much sharper upper bound on the maximum roundoff error due to encoding. Since m does not depend on n, these strong numerical properties scale well to larger systems.

Since the upper bound on roundoff error due to encoding is proportional to the Euclidean norm of the encoder vector, the partitioned encoding scheme improves the error bound by a factor of

$$\gamma = \frac{\|g_n^T\|_2}{\|g_m^T\|_2}$$

where g_n^T is the encoder vector of length n and g_m^T is the corresponding partitioned encoder vector of length m. As a result, the value of Δ, the maximum error tolerated, can be comfortably decreased by this same factor γ. Thus, the partitioned scheme improves fault coverage without increasing the possibility of *false alarms*. In particular, a fault that results in a code error magnitude between and Δ/γ and Δ can be detected using the partitioned encoding scheme, whereas the traditional coding method would not detect the fault.

The roundoff error E also depends on $\|x\|_2$, the norm of the vector to be encoded. In general, this vector has length n; however, in the partitioned scheme, x has only m elements. Any m-element partition of the length n vector always has norm less than or equal to that of the n-element vector. This further serves to decrease the roundoff error due to the encoding and improve fault coverage.

The value of γ depends largely on the structure of the encoder vector g^T. For the weighted checksum vector e_w^T, the error bound is improved by a factor of $\gamma = 2^{n-m}$, which can be quite significant. In the case of the unweighted checksum vector e^T, $\gamma = \sqrt{n/m}$. As n grows, the factor γ plays an important role in maintaining reasonable worst-case error thresholds. This is crucial to achieving acceptable fault coverage in a massively parallel system.

4 Conclusion

The method suggested in this paper can be applied to any general linear code, in order to extend the applicability of ABFT techniques. The partitioned coding scheme requires a small time and hardware overhead that scales well as n grows. For this cost, the partitioned approach prevents overflow in encoding, and alleviates the problem with the reflectivity of codes, while guarding against roundoff error in encoding. The sharper bound on numerical encoding error allows the method to provide more complete fault coverage. The scheme provides a general, low-cost way to scale effective ABFT techniques to massively parallel floating-point systems.

References

[1] K.-H. Huang and J. A. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE Trans. Comput.*, vol. C-33, pp. 518-528, June 1984.

[2] J.-Y. Jou and J. A. Abraham, "Fault tolerant matrix arithmetic and signal processing on highly concurrent computing structures," *Proc. IEEE*, vol. 74, no. 5, pp. 732-741, May 1986.

[3] F. T. Luk and H. Park, "An analysis of algorithm-based fault tolerance techniques," in *Proc. SPIE Adv. Alg. & Arch. for Signal Proc.*, vol. 696, pp. 222-228, Aug. 1986.

[4] A. L. N. Reddy and P. Banerjee, "Algorithm-based fault detection for signal processing applications," *IEEE Trans. Comput.*, vol. 39, pp. 1304-1308, Oct. 1990.

[5] B. Vinnakota and N. K. Jha, "A dependence graph-based approach to the design of algorithm-based fault tolerant systems," in *Proc. Int. Symp. Fault Tolerant Comput.*, Newcastle-upon-Tyne, U.K., pp. 122-129, June 1990.

[6] J.-Y. Jou and J. A. Abraham, "Fault tolerant FFT networks," *IEEE Trans. Comput.*, vol. 37, no. 5, pp. 548-561, May 1988.

[7] Y.-H. Choi and M. Malek, "A fault tolerant FFT processor," *IEEE Trans. Comput.*, vol. 37, no. 5, pp. 617-621, May 1988.

[8] S.-J. Wang and N. K. Jha, "Algorithm-based fault tolerance for FFT networks," in *Proc. Int. Symp. Circuits & Systems*, San Diego, CA, May 1992.

[9] Y.-H. Choi and M. Malek, "A fault tolerant systolic sorter," *IEEE Trans. Comput.*, vol. 37, no. 5, pp. 621-624, May 1988.

[10] P. Banerjee and J. A. Abraham, "Bounds on algorithm-based fault tolerance in multiple processor systems," *IEEE Trans. Comput.*, vol. C-35, no. 4, pp. 296-306, Apr. 1986.

[11] V. S. S. Nair and J. A. Abraham, "A model for the analysis of fault tolerant signal processing architectures," in *Proc. 32nd Int. Tech. Symp. of SPIE*, San Diego, pp. 246-257, Aug. 1988.

[12] B. Vinnakota and N. K. Jha, "Diagnosability and diagnosis of algorithm-based fault tolerant systems," accepted for publication in *IEEE Trans. Comput.*

[13] C. J. Anfinson and F. T. Luk, "A linear algebraic model of algorithm-based fault tolerance," *IEEE Trans. Comput.*, vol. 37, no. 12, pp. 1599-1604, Dec. 1988.

[14] V. S. S. Nair and J. A. Abraham, "General linear codes for fault tolerant matrix operations on processor arrays," in *Proc. Int. Symp. Fault-Tolerant Comput.*, Tokyo, pp. 180-185, June 1988.

[15] W. Bliss, M. Lightner, and B. Friedlander, "Numerical properties of algorithm-based fault tolerance for high reliability array processors," in *Proc. 22nd Asilomar Conf. Signals, Systems, & Computers*, vol. 1, pp. 631-635, 1988.

[16] F. Luk, "Algorithm-based fault tolerance for parallel matrix equation solvers," SPIE vol. 564, *Real Time Signal Processing VIII*, pp. 49-53, 1985.

[17] J. A. Abraham et al., "Fault tolerance techniques for systolic arrays," *IEEE Computer*, pp. 65-74, July 1987.

[18] D. J. Rosenkrantz and S. S. Ravi, "Improved upper bounds for algorithm-based fault tolerance," in *Proc. 26th Allerton Conf. Comm. Cont. & Comput.*, Allerton, IL, pp. 388-397, Sept. 1988.

[19] P. Banerjee et al., "An evaluation of system-level fault tolerance on the Intel hypercube multiprocessor," in *Proc. Int. Symp. Fault Tolerant Comput.*, Tokyo, pp. 362-367, June 1988.

[20] V. S. S. Nair and J. A. Abraham, "A model for the analysis, design and comparison of fault-tolerant WSI architectures," in *Proc. Workshop Wafer Scale Integration*, Como, Italy, June 1989.

[21] V. S. S. Nair and J. A. Abraham, "Hierarchical design and analysis of fault-tolerant multiprocessor systems using concurrent error detection," in *Proc. Int. Symp. Fault Tolerant Comput.*, Newcastle-upon-Tyne, U.K., pp. 130-137, June 1990.

[22] D. Gu, D. J. Rosenkrantz, and S. S. Ravi, "Design and analysis of test schemes for algorithm-based fault tolerance," in *Proc. Int. Symp. Fault Tolerant Comput.*, Newcastle-upon-Tyne, U.K., pp. 106-113, June 1990.

[23] A. M. Cohen, *Numerical Analysis.* New York: John Wiley and Sons, 1973.