ELSEVIER

Microprocessors and Microsystems 21 (1997) 151–161

MICROPROCESSORS AND MICROSYSTEMS

Algorithm-based fault tolerance: a review

M. Vijay*, R. Mittal

Department of Computer Science and Engineering, Indian Institute of Technology, Madras 600 036, India

Abstract

The need for reliability of computers has been increasing, as computers have been put to use in more and more practical applications. Multiprocessor architectures have provided elegant solutions for certain computationally expensive problems which find wide-ranging applications in areas such as defense and industry. Since computer-intensive applications are run on these architectures, the probability that some computations will incur error is not negligible. Hence fault tolerance plays an important role in the design of multiprocessor architectures. In this paper, we review a low-cost scheme for adding fault tolerance in multiprocessor architectures, called algorithm-based fault tolerance (ABFT). The concurrent error detecting and correcting capabilities of this scheme are demonstrated with the help of examples. Various issues of interest, the areas open to research and the limitations of ABFT are also pointed out. © 1997 Elsevier Science B.V.

Keywords: Algorithm-based fault tolerance; Fault tolerance; Multiprocessor architectures; Space redundancy; Time redundancy

1. Introduction

VLSI technology has paved the way for the development of massively parallel computational systems. In these systems, the probability of any one of the large number of processing elements failing is quite high. In addition, there is significant increase in the number of transient errors due to rapidly decreasing geometries of digital circuits.

Furthermore, the demand for accurate and precise computation has also risen. This is mainly because computers are increasingly being used in applications where an error in the computation is costlier than that of the equipment. The use of computers in defense and avionics is evidence supporting this fact. All these factors have increased the need for fault tolerance in computer systems.

Many techniques for adding fault tolerance have been proposed in the literature [1–5]. The fault tolerance techniques based on error masking and reconfiguration use time redundancy and space redundancy. These techniques use a significant amount of resources to detect the faulty components, mask the faults, and reconfigure the system. Offline testing techniques cannot be applied to many real-time systems as these techniques are time consuming and often incomplete.

Algorithm-based fault tolerance (ABFT) is an error detection, location and correction scheme which uses redundant computations within the algorithms to detect and correct errors caused by permanent or transient failures in the hardware, concurrently with normal operation [6]. In this paper, we review the ABFT schemes for adding fault tolerance. We study their efficiency over the traditional schemes. For certain specific examples, we compare the hardware and time overheads in the ABFT scheme to that of pure time redundancy and space redundancy-based schemes. Matrix operations are one of the most widely encountered computations in practical applications. In the remainder of this paper we concentrate on the applications of ABFT to matrix operations.

The rest of the paper is organized as follows. In Section 2, we look at the origin of ABFT and the motivation behind the scheme. The fundamental concept behind ABFT, the fault model for a general ABFT scheme and a few preliminary terminologies and parameters of an ABFT system are discussed in Section 3. In Section 4 we examine certain examples of ABFT systems and give a comparison of the overheads involved with respect to the other fault tolerant schemes. We also look at the role of architecture in designing an ABFT system. We discuss the theoretical aspects of ABFT in Section 5. In Section 6 we point out the limitations of ABFT schemes. Finally, a summary of our observations is given in Section 7.

2. Origin of ABFT

One way of classifying fault tolerant schemes is time redundancy and space redundancy. Time redundancy is a popular technique used for adding fault tolerance, in which the same computation is performed on the same processor at

* Corresponding author.

PII S0141-9331(97)00029-X

two (or more) different but close enough time periods, and then the results are compared. In space redundancy, each computation is done at more than one location in space (on different processors) and the results are compared. Generally, these computations are performed concurrently. Another dimension in the classification of fault tolerant techniques is hardware and software fault tolerance. These two dimensions together constitute four different models of fault tolerance, as depicted in Table 1.

All the above fault-tolerant techniques contain the following stages:

- error detection;
- fault location;
- reconfiguration; and
- recovery and continued service.The above stages of fault tolerance apply well to all the four classes of fault tolerant schemes we have discussed above. An error can be detected, for example, by a mismatch in the compared values. Then the faulty component is located. The faulty component has to be isolated so that the system can continue operations after reconfiguration. Since an error has occurred in the computation, the computation has to be repeated (in certain cases), to obtain the required error-free output, and this constitutes recovery. All these stages might not be present with the same rigour in all fault tolerant schemes. For example, in N-version programming, if a mismatch is detected in the N outputs from the N versions, the majority will be voted for the required output, and this does not involve any separate recovery.

In 1984, Huang and Abraham [6] investigated different techniques of fault tolerance in which the space and time overheads would be minimized. The idea of checksums led them to develop a new fault tolerant technique called algorithm-based fault tolerance (ABFT). They applied this technique to certain matrix operations, and their ABFT mechanism showed a clear superiority over the existing techniques of fault tolerance with respect to space and time overheads. Furthermore, no reconfiguration and recovery were required by this mechanism. ABFT became popular as a concurrent error detection/error correction mechanism, and has recently been widely applied to several computational problems [7–11].

3. Preliminaries

It should be noted that ABFT was originally designed with multiprocessor architecture in mind. The matrix operations that were attempted by Huang and Abraham [6] were implemented on an array of processing elements (*PE*s). As described in the the fault model, we will concentrate on the faults in the *PE*s in a multiprocessor configuration. We assume that the algorithm is to be executed on a set of *PE*s with each performing a part of the entire computation. The exact interconnection pattern between these *PE*s will be taken into account when we study the examples in detail.

Table 1

	Space redundancy	Time redundancy
Software	N-version programming	Retry block approach
Hardware	N-modular redundancy	Triple time redundancy

3.1. Basic idea

In an ABFT system, the input data is encoded. The algorithm is modified to work on the modified input. The computation steps of the algorithm are then distributed among the computation units. It is important to allocate the computational steps on different processors so that an error in one processor affects as few data elements as possible. The encoded output produced by the algorithm is decoded to check the correctness of the results. The following scheme summarizes the operations in an ABFT system:

$$\text{Input} \rightarrow \text{Encoded input} \xrightarrow[\text{algorithm}]{\text{Modified}} \text{Encoded output}$$

$$\rightarrow \text{Decoded output}$$

An ABFT system uses redundant computations within the algorithm to detect and correct errors caused by permanent or transient failures in the hardware. From this, it is clear that this approach is not a general mechanism as other approaches (e.g. the triple modular redundancy). On the contrary, it varies from algorithm to algorithm. However, when the modified algorithm is actually executed on a multiprocessor architecture, the overheads are required to be minimum in comparison to TMR. The modified algorithm could, of course, take more time to operate on the encoded data when compared to the original algorithm.

In some of the existing fault tolerant schemes, checks are performed on the *PE*s to test the correctness of the output data. In an ABFT scheme, checks are always performed on the data elements produced by the *PE*s. For this to be possible, the faulty *PE* must not mask the error during the error detection or error correction phase.

3.2. Basic fault model

ABFT techniques generally address fault tolerance *within* a processing element (*PE*). We consider a module-level fault model, in which a module (a *PE* of a multiprocessor) makes arbitrary logical errors in the event of a fault. Since transient errors have been reported more often, we assume that permanent errors are rare in these systems. A fault in a system is a malfunction of either the software or the hardware constituting the system, and an error is a manifestation of that fault in the computations, or in the data elements (output). The fault can be a computational fault or an

addressing fault. We note that a fault in the system may not always lead to an error, but an error will always be caused by some fault. We do not distinguish the case in which a fault does not manifest itself as an error, from the fault-free case. A graph theoretical method to model these faults is presented in Section 5.

3.3. Basic definitions

1. For a processor PE_i, the data set $DS(i)$ is the set of data elements affected by PE_i.
2. An ABFT system has a set of checks $C_1, C_2, \ldots, C_p$ which checks the correctness of the data elements produced by the PEs. The error set $ES(k)$ of a check C_k is the set of data elements checked by C_k. The cardinality of $ES(k)$ is bounded. The maximum number of errors in data elements which the check C_k can handle without giving an incorrect decision is also bounded.
3. A check in an ABFT system is a (g, h) check if:
4. the cardinality of its error set is g;
5. the check passes (outputs 0) if none of the data items is in error;
6. the check fails (outputs 1) if no more than h elements in its error set are erroneous; and
7. the state of the check is unpredictable if more than h elements in its error set are erroneous (assuming that the check-computing processor is non-faulty).
8. An ABFT system is said to be s-error detectable if the presence of s or fewer erroneous data elements in the system is indicated by at least one check in the system going to state 1.
9. An ABFT system is said to be s-error locatable if, given s or fewer erroneous data elements, the PEs which resulted in these errors can be located from the state of the checks in the system.

4. Applications of ABFT

4.1. ABFT in matrix operations

If $\mathbf{A}$ is an $n \times n$ matrix, a row checksum matrix $\mathbf{A}_r$ is a $n \times (n+1)$ matrix, as defined below:

$$\mathbf{A}_r = [\mathbf{A}\ \mathbf{Ae}]$$

where $\mathbf{e}$ is an $n \times 1$ vector given by

$$\mathbf{e} = [1\ 1 \ldots 1]^T$$

By comparing $\sum_{j=1}^{n} a_{ij}$ and $(\mathbf{Ae})_i$, $i = 1,2,\ldots, n$, we can detect an error in the ith row. Similarly, a column matrix $\mathbf{A}_c$ is defined as

$$\mathbf{A}_c = \begin{bmatrix} \mathbf{A} \\ \mathbf{A}\mathbf{e}^T \end{bmatrix}$$

where $\mathbf{e}^T\mathbf{A}$ is called the column summation vector. By comparing $\sum_{i=1}^{n} a_{ij}$ and $(\mathbf{e}^T\mathbf{A})_j$, $j = 1, 2, \ldots, n$, we can detect an error in the jth column. From the full checksum matrix,

$$\mathbf{A}_f = \begin{bmatrix} \mathbf{A} & \mathbf{Ae} \\ \mathbf{e}^T\mathbf{A} & \mathbf{e}^T\mathbf{Ae} \end{bmatrix}$$

we can determine the exact location of one error in the matrix by intersecting inconsistent rows and columns. This can also be used to correct a single error, using the inconsistent row or column. Each row/column in $\mathbf{A}_f$ is called the CSEV (checksum encoded vector).

Huang and Abraham [6] suggested this checksum method for some basic matrix operations such as matrix addition, matrix multiplication, scalar product, transposition, and LU decomposition. They mentioned that the checksum method can be applied to any matrix operation for providing fault tolerance only if the checksum property can be ensured within the computation. The *checksum property* was defined as the condition in which the checksum elements always hold the value of the summation of their corresponding rows or columns. For example, all the matrix operations mentioned above preserve the checksum property. When any of these operations are carried out between two full checksum matrices, the resultant matrix will also be a full checksum matrix. Only those operations which preserve this property are used in the computations.

Example 1. Two full checksum matrices $\mathbf{A}$ and $\mathbf{B}$ are given below:

$$\mathbf{A}_f = \begin{pmatrix} 1 & 2 & 3 & 6 \\ 2 & 3 & 4 & 9 \\ 3 & 4 & 5 & 12 \\ 6 & 9 & 12 & 27 \end{pmatrix}, \quad \mathbf{B}_f = \begin{pmatrix} 1 & 0 & 1 & 2 \\ 2 & 1 & 0 & 3 \\ 0 & 3 & 0 & 3 \\ 3 & 4 & 1 & 8 \end{pmatrix}$$

Matrices $\mathbf{A}$ and $\mathbf{B}$ which were 3×3 matrices were made full checksum matrices, by adding an extra row and column of elements, in each of them. It can be observed that the full checksum matrices $\mathbf{A}_f$ and $\mathbf{B}_f$ satisfy the checksum property. Matrix $\mathbf{C}_f$ which is obtained by multiplying $\mathbf{A}_f$ and $\mathbf{B}_f$ is shown below:

$$\mathbf{C}_f = \mathbf{A}_f \times \mathbf{B}_f = \begin{pmatrix} 23 & 35 & 7 & 65 \\ 35 & 51 & 11 & 97 \\ 47 & 67 & 15 & 129 \\ 105 & 153 & 33 & 291 \end{pmatrix}$$

The last row (last column) in $\mathbf{C}_f$ contains values which are sums of the elements of the corresponding columns (rows). Hence $\mathbf{C}_f$ also satisfies the checksum property. This example demonstrates that the operation of multiplication of matrices preserves the checksum property.

When the matrix elements are stored as floating point numbers, the checksum technique does not significantly

affect the word length. For example, even for $n = 200$, a checksum element may be up to 4×10^4 times the maximum number in the information matrix; this is, however, only an increase of four in the exponent for base 16 systems and 16 for base 2 systems. When processing matrices with integer elements, the same word length r can be used to store the elements of the summation vectors if residue arithmetic is applied to the computation vectors with a modulus $M = 2^r$.

The use of either row checksum or column checksum matrices makes the system 1-error detectable. A full checksum matrix system is also 1-error correctable. It was shown that only small redundancy ratios—$O(1/n)$ for hardware and $O(\lg n/n)$ for time—are required for processor array systems to achieve fault-tolerant matrix operations. The same technique is also modified to suit applications involving large matrices with a limited number of processors [6]. A matrix is partitioned into several sub-matrices and each sub-matrix can be encoded to fit the size of a given processor array. In each checksum sub-matrix there is, at most, one erroneous element. Thus the error can be detected and corrected. It was also proved in Ref. [6] that at least $n^2/(2 \times n - 1)$, $(n \geq 2)$ processors are required to produce an $n \times n$ checksum matrix in which errors caused by a faulty processor can be detected.

To increase the error correcting capability of the system, Jou and Abraham [11] proposed a *weighted checksum approach*. With the introduction of d weighted checksum rows (or columns), for a suitable choice of weight vectors, we can make the system d-error detectable and $\lfloor d/2 \rfloor$-error correctable. The weighted row-checksum matrix is defined as

$$\mathbf{A}_{\mathrm{rw}} = [\mathbf{A}\ \mathbf{A}\mathbf{w}^{(1)}\ \mathbf{A}\mathbf{w}^{(2)} \ldots \mathbf{A}\mathbf{w}^{(d)}]$$

In the case when $d = 2$, where $\mathbf{w}^{(1)} = \mathbf{e}$ and $\mathbf{w}^{(2)} = \mathbf{w} = [1\ 2\ 3 \ldots n]$, if the erroneous element is say, a_{pq}, then the possibly erroneous version of a_{ij} is denoted as a'_{ij}, for $1 \leq i, j \leq n$. Define

$$s_1 = \sum_{k=1}^{n} a'_{kq} - (\mathbf{e}^{\mathrm{T}}\mathbf{A})_q$$

$$s_2 = \sum_{k=1}^{n} w_k a'_{kq} - (\mathbf{w}^{\mathrm{T}}\mathbf{A})_q$$

Examining $s_2/s_1 = p$, we can exactly locate the error in the (p,q) position of A. To correct it we use

$$a_{pq} \leftarrow a'_{pq} + s_1$$

Example 2. Consider the following weighted checksum matrix, which was obtained as the output of some operations on other weighted checksum matrices:

$$\mathbf{A}_{\mathrm{rw}} = \begin{pmatrix} 1 & 2 & 3 & 5 \\ 2 & 3 & 7 & 10 \\ 3 & 0 & 3 & 3 \end{pmatrix}$$

$\mathbf{A}$ is originally a 2×2 matrix. The extra two columns are added to make it a weighted checksum matrix. Here, $\mathbf{w}^{(1)}$ is taken as $[1\ 1]^{\mathrm{T}}$, and $\mathbf{w}^{(2)}$ as $[1\ 2]^{\mathrm{T}}$. It can be seen that the erroneous element lies in the second row, since the checksum elements in that row do not satisfy the checksum property. From the notation described above, $s_1 = 2$ and $s_2 = 2$. Since $s2/s1 = 1$, the erroneous element should be present in the first column. Hence $\mathbf{A}_{\mathrm{rw}}(2,1)$ is the erroneous element. It can be corrected by setting $\mathbf{A}_{\mathrm{rw}}(2,1)$ to $2 + s_1$, which is 4.

Weighted checksums have been found to be unstable in their floating point and fixed point arithmetic. A tolerance level must be fixed to take into account the rounding errors which arise during the computation. Another problem that occurs with the floating point operations is caused by overflow due to extremely large weights. For example Jou and Abraham [11] suggested $w_j^{(i)} = 2^{(j-1)(i-1)}$. This choice will easily result in an overflow. One suggestion [12] has been to let $w_j^{(i)} = j$. Modulo arithmetic can be used to compute weighted checksums, but this has the disadvantage of complicating the hardware. It also makes the fault location more difficult since the difference between successive weights is small. Additionally, an accumulated error of ± 0.5 results in non-unique weights. An optimal choice of weight vectors taking into account practical problems into consideration is still a topic of research.

Other matrix operations were attempted by some researchers. Matrix inversion with maximum pivoting was solved by Yeh and Feng using checksum methods [13]. A series of row and column operations were defined in this work which satisfy the checksum property.

A *partitioned encoding scheme* was suggested by Redford and Jha [14] for massively parallel systems. Instead of having one checksum row (or column) for the entire matrix, they suggested dividing the matrix into sub-matrices and having row checksums (or column checksums) for each sub-matrix. They proved that this method will be more scalable and resistant to floating point problems mentioned earlier (thus preventing false alarms.)

Adaptive pivoting is another new topic in this area. Pivoting of a matrix involves the multiplication of a row/column by a factor k so that an upper or lower triangular matrix is obtained by subtracting the row/column from another row/column containing the pivot. Khan et al. [15] showed that the sequence of selecting the pivots in successive phases of pivoting does not affect the final result. They also applied a relaxation as follows: Any n elements can be used as a pivot as long as no two are used are from the same row or column. However, this relaxation has an important side-effect of changing the mapping of the final elements. This is overcome by retracing the the convoluted mapping through a distributed retracing scheme using the phase sequence number and corresponding pivot position which are available locally to the processor nodes. As a result, the entire mapping can be retraced without any extra communication.

Adaptive pivoting makes prudent use of the above relaxation, and instead of interchanging rows, it adaptively skips pivot positions depending on their values (zero or too small a value suggests skipping).

4.2. ABFT in FFT networks

The discrete Fourier transform is one of the most important computations in digital signal processing [16]. It is used in computing the convolution of two signals in many applications. The discrete Fourier transform is given by

$$X(k) = \sum_{n=0}^{N-1} \mathbf{x}(n) w_N^{kn}, \quad k = 0, 1, \ldots, N-1$$

where w_N $e^{-j(2\pi/N)}$ is the Nth root of unity and is called the twiddle factor. It is possible to rewrite the above equations as

$$X(k) = X_{\text{even}}(k) + W^k X_{\text{odd}}(k), \quad 0 \le k \le N/2 - 1$$

$$X(k + N/2) = X_{\text{even}}(k) - W^k X_{\text{odd}}(k), \quad 0 \le k \le N/2 - 1$$

where X_{even} is the Fourier transform of the even points of X, and X_{odd} is the Fourier transform of odd points of X.

We now describe the detection and location of faulty processors concurrently with the FFT computations on a hypercube multiprocessor, using an ABFT scheme [16]. Here we assume the following:

1. The number of points is a power of 2, e.g. 2^p.
2. The number of processors (processing nodes) is a power of 2, e.g. 2^N. Note that the size of a hypercube is always a power of 2.
3. The number of points is more than or equal to the number of nodes.The parallel algorithm [16] is described below.

(a) The host computer will divide the points evenly among the nodes. If there are 2^P number of points, and there are 2^N number of nodes, then each node will receive 2^{P-N} points.
(b) The points are sent to the different nodes in permuted manner. Let y be the bit-reversal-permutation of of the input vector **x**. Send points to different nodes as given below.
for $i = 0$ **to** $2^N - 1$
send $y[i \times 2P - N \cdots (i+1) \times 2^{P-N}]$ to node i.
endfor.
Alternatively, all the points are broadcast to all the nodes and each node keeps all points it needs
(c) Each processing node will parallely perform the iterative sequential FFT algorithm on the points they have received. Each node performs computations independently.
(d) After completion of FFT computations of the received points, each node exchanges message as given below:
for $i = 0$ **to** $N - 1$

neighbour = complement bit i of id
if (*id* < *neighbour*)

Exchange the second half of node *id* with the first half of the neighbour

else

Exchange the first half of node *id* with the second half

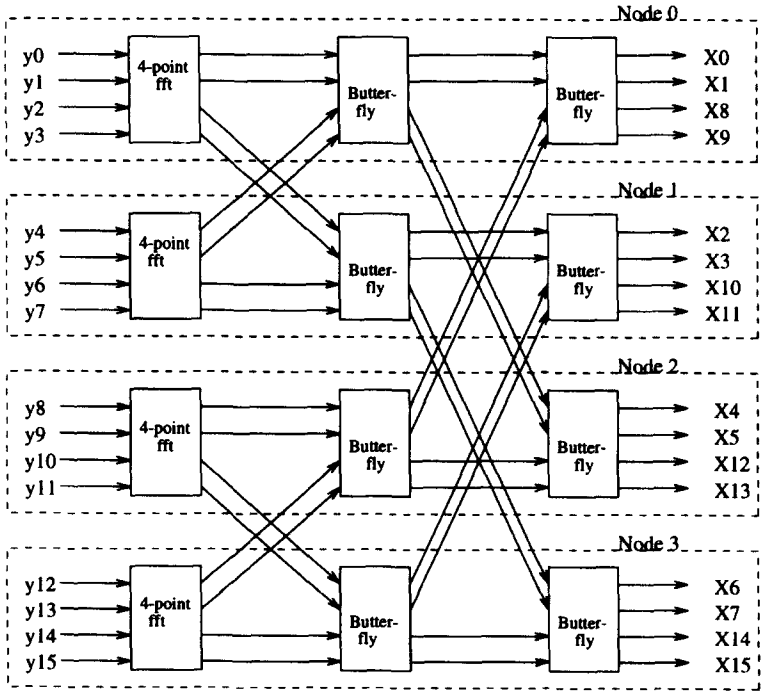


Fig. 1. Sixteen point FFT on four-processor hypercube.

of the neighbour

Perform the butterfly operation

endfor

(e) Each node will pass the results back to the host.

The working of a 16-point FFT algorithm on four nodes is illustrated in Fig. 1.

The two properties of the FFT which are used as checks in the hypercube algorithm [16] are described below:

Property 1. The product of the number of points in the FFT and sum-of-squares of the input points equals the sum-of-squares of the output points [16].

Property 2. The results of intermediate computations in the parallel algorithm for the FFT are related by the equality of twice the sum of squares of the outputs of intermediate FFT computations. In Fig. 1 the results of the sum of squares of inputs to an eight-point FFT computation given the four-point FFTs of the even and odd points equals half the sum of squares of the outputs of the eight-point FFT computation given the four-point FFT's of the even and odd points equals half the sum of squares of the outputs of the eight-point FFT.

The error detection mechanism [16] is as follows:

1. Each node computes the sum of squares (SOS) of its input and output points and sends the results to node 0 and 1, respectively. For example node 2 computes SOS(y8, y9, y10, y11) and sends the result to node 0. It also computes SOS(X4, X5, X12, X13) and sends the result to node 1.
2. Nodes 0 and 1 compute the total SOS of the input and output, respectively, from the partial SOS they receive from the other nodes.
3. Node 0 obtains the outpout SOS from node 1, and node 1 obtains the input SOS from node 0.
4. Both nodes 0 and 1 check the computation by computing SOS(output) − SOS(input)/N and compare it to a tolerance factor. The results of comparison, *pass* or *fail*, are sent to the host.
5. The host declares the computation to be erroneous if it receives a *fail* indication from either node 0 or 1. Some additional checks are performed to locate the faulty processor assuming the fault to be permanent.The results of intermediate computations are saved as checkpoints among the processing node in our parallel algorithm. These intermediate results are checked using the same property, e.g. the sum of squares of x(0), x(4), x(8), and x(12), multiplied by 4, should equal the sum of squares of the four-point FFTs computed by node 0. These computations are checked by node 0 itself and its two neighbours: nodes 1 and 2. The results of the check are sent to the host. As the results are computed by three processors independently, the faulty processor can be identified by the host if the fault appeared in this stage of the computation.

In the next step the correctness of the second stage of computations (eight-point FFTs) is verified. The inputs and outputs of a portion of the eight-point FFT (points 0, 1, 4, 5) are sent by node 0 to two neighbours: nodes 1 and 2. Nodes 0, 1 and 2 independently compute the sum of squares and send three set of results to the host. The host verifies whether the sum of squares of the inputs at this stage multiplied by two equals the sum of squares of the outputs. If an error is determined at this stage, the faulty processor is located to within two processors, e.g. nodes 0 and 1, or node 2 and 3.

The next step verifies the correctness of the third stage (16-point FFT). In this stage, inputs and outputs of the points 0, 1, 8 and 9 are sent from node 0 to its two neighbours. Again, nodes perform the intermediate sum of squares of inputs and outputs and send three sets of results to the host. The host checks for the equality of twice the sum of squares of the inputs to the sum of squares of the output (property 2). Any difference at this stage can locate the faulty processor to within four processors i.e. the entire hypercube.

Now the host assumes that the fault is transient and repeats the entire set of computations. In case the same syndrome is repeated, the fault is assumed to be permanent.

Some of the other fault tolerant schemes for FFT networks are as follows:

(1) Choi and Malek [17] proposed a fault tolerant scheme for FFT, based on recomputation through an alternate path. The throughput of this system is only 50% compared to a system with no fault tolerance.

(2) Jou and Abraham [18] proposed a *checksum embedded design* (CED) combining the original DFT with a rotated DFT. The checksum comparison circuit was realized using an inverse DFT (see Fig. 2). The hardware overhead of this scheme is approximately 2/lg N for an N-point FFT only if the multipliers are counted in considering the hardware complexity. Due to round-off errors, this scheme's fault coverage, throughput or both may not be satisfactory.

(3) Reddy and Banerjee [19] presented an encoding scheme of sum of squares of input and output data. The hardware complexity was 75% of that of Jou and

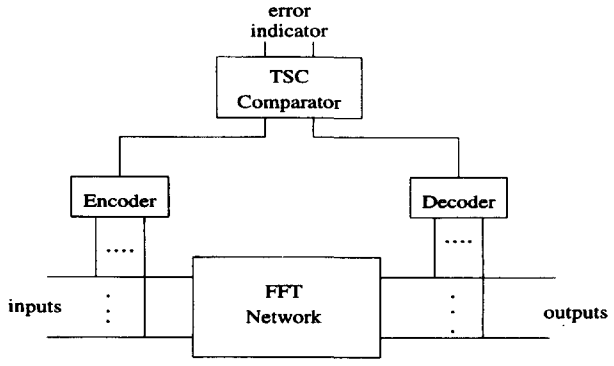


Fig. 2. A general CED scheme.

Abraham for real input, but the fault model was much weaker.

(4) Wang and Jha [8] proposed a new weighted CED scheme. The input encoder generates input weighted check-sum

$$WCS_i = \overline{\mathbf{ws}} \times \bar{\mathbf{x}}^T$$

and the output decoder computes

$$WCS_o = \bar{\mathbf{r}}_w \times \bar{\mathbf{X}}^T$$

where $\bar{\mathbf{r}}_w$ is the row weight vector, $\bar{\mathbf{x}}$ is the input vector, $\bar{\mathbf{X}}$ is the output vector and $\overline{\mathbf{ws}}$ is an encoding vector. WCS_i and WCS_o are then compared by the TSC comparator to decide if they are equal. They showed that every single functional fault in an FFT network is detectable. They also showed that the fault coverage of this scheme is less sensitive to an increase in the network size.

(5) Tao and Hartmann [20] later reduced the number of multipliers by slightly modifying the process for the output checksum calculation. Thus the hardware overhead was reduced to 1.8/lg N.

(6) Oh et al. [9] proposed a simple linear weight factor-based checksum approach. A distinct weight factor $F_k = k + 1/N$ is used. Since the weight factor is linear and small (≤ 1), no overflow is possible and also the error masking probability is reduced because of the regularity and symmetry of the FFT network. This also implies reduced hardware overhead and small error detection latency. The scheme can detect multiple errors unless they compensate each other. In addition, the output checksum is evaluated without using multipliers. All the above factors contribute to reduce the hardware overhead to 1.2/g N for real input and 2/lg N for complex input.

4.3. ABFT and systolic ARRAYS

As mentioned earlier, ABFT schemes were designed mainly for the multiprocessor architecture. In this sub-section, we examine the role of the architecture in designing an ABFT mechanism. We consider the example of systolic arrays to study this.

Systolic arrays, provide elegant solutions to problems which are compute-bound, without changing the I/O requirements [21,22]. They have the following advantages:

1. The *PE*s which constitute the systolic array are simple and identical. Additionally, the interconnection between the *PE*s are symmetric and local. These factors facilitate easy fabrication on a chip using the VLSI technology.
2. The required data is input to a systolic array at the boundary *PE*s. The intermediate results are transfered from *PE* to *PE* within the systolic array, and since the interconnections are local, these data transfers take less amount of time compared to the time taken for transferring the intermediate results to temporary storage and back when required. This gives a high throughput of the results.
3. Systolic arrays use temporal parallelism (pipelining) in the algorithm.

The design of a systolic array for a particular problem involves the following steps:

1. Obtain a space–time recursive formulation of the problem. One of the common ways to describe an algorithm is to use recursive equations. A space–time recursive formulation is just a recursive equation in which one of the indices is used to represent space, and the other is used to represent time.
2. Obtain the *dependency graph* (DG), from the space–time recursive formulation (see Fig. 3). A DG is a graph that shows the dependence between the various computations in the algorithm. Each node in the DG denotes a computation, and a directed arc between two nodes indicates the precedence relation between the corresponding computations. An algorithm is computable if and only if its complete DG has no cycles or loops. Although many methods have been proposed to construct a DG from sequential code, a formal and automatic methodology remains a major open research problem. An extraction of the partial ordering of operations in an algorithm will yield a DG expression.
3. Map the nodes of the DG onto an array of *PE*s, taking into account the following points, to obtain the systolic array:
4. In order to improve the *PE* utilization, it is often desirable to map the nodes of the DG onto a fewer number of *PE*s.
5. Each node in the DG has to be mapped to a *PE* of the systolic array.
6. *PE* can perform only one computation at any given point of time. So, computations which are to be performed at the same time should not be mapped to the same *PE*.
7. If there is a precedence relation from node N_1 to node N_2 in the DG, then the computation corresponding to N_1 has to be done on a *PE* (say PE_1) before the computation corresponding to N_2 is done on a *PE* (say PE_2). In addition, PE_1 and PE_2 have to be either the same or adjacent, so that the results of the computations at PE_1 are easily accessible to PE_2.

Vinnakota and Jha [23] have proposed a DG-based approach for adding fault tolerance to systolic arrays. They considered the problem of FIR filters to demonstrate their approach. The convolution between $X(i)$, $1 \leq i \leq n$, and a sequence of coefficients $B(i)$, $1 \leq i \leq q$ is given by

$$Y(q) = \sum_{i=0}^{q} X(i-q)B(i)$$

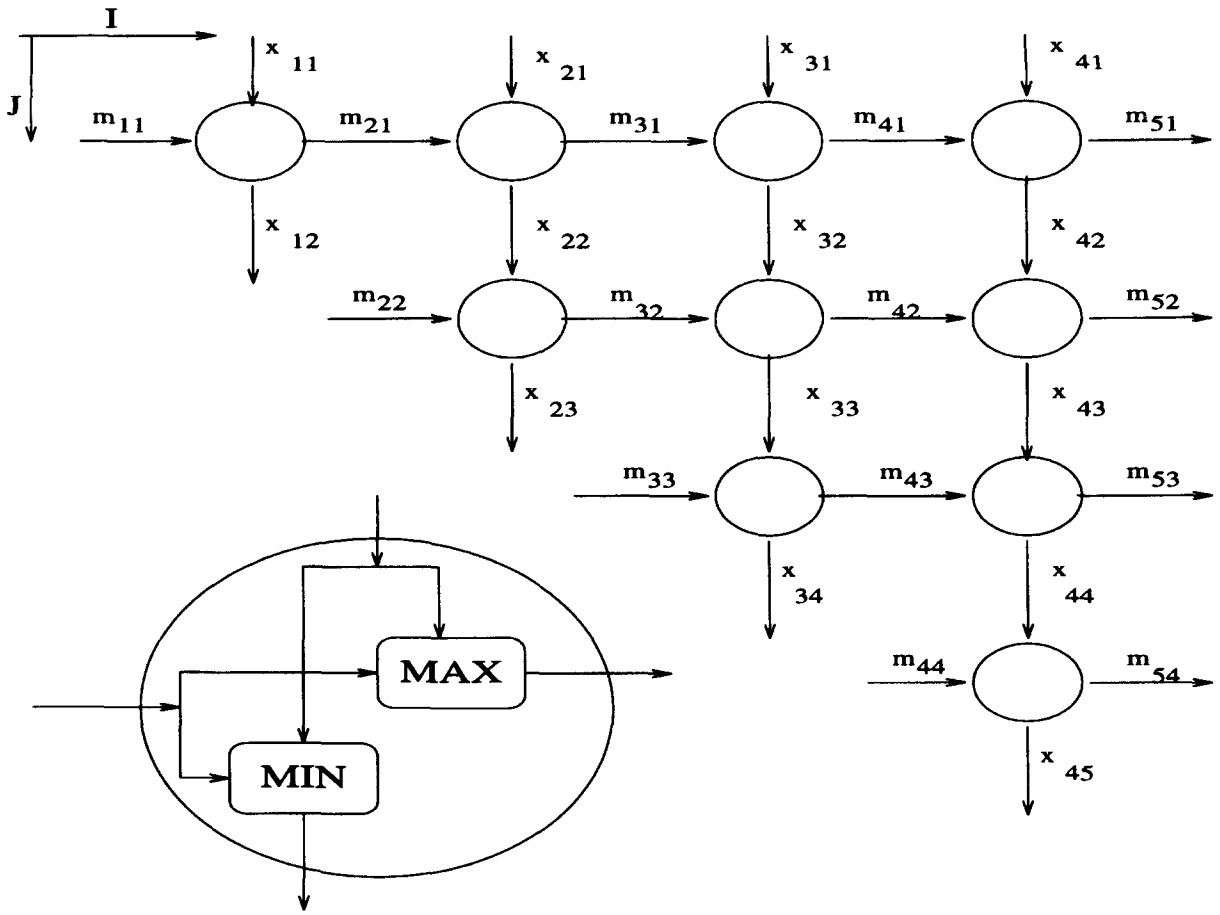


Fig. 3. DG for sorting. The original sequence $x(i)$ is input from the upper row as $x(i,1)$. The data $m(i,i)$ along the diagonal nodes is set to be $-\infty$. The sorted sequence $m(j)$ is output from the rightmost column as $\{m(N, j), j = 1, 2, \ldots, N\}$.

The DG for this computation is modified so that extra computations are performed by the modified DG along with the required ones. The model proposed in this paper could only detect errors. The method of correction of the errors is not discussed. A rollback is required in such schemes once an error is detected. The error that occurs at a *PE* is propagated to the rest of the *PE*s, and this results in a greater number of errors. Hence it is desirable that the errors in an ABFT system are localized.

Vijay and Murthy [24] have extended this DG-based approach to the evaluation of polynomials and exponentials of polynomials at equispaced arguments. Their approach has the property of localization of errors, i.e. the errors will be corrected by the neighbouring *PE*s, as soon as they are detected and they are not propagated to other *PE*s for further computations. DGs were used not only to add fault tolerance, but also to study the error detectability and correctability capabilities, in the latter work. The DG-based ABFT technique in Ref. [24] does not require any roll back, and hence provides a time-conservative solution for the problem considered. In the same work, Vijay and Murthy also considered time-redundancy techniques to add fault tolerance to the system which evaluates polynomials and exponentials of polynomials at equispaced arguments. The time-redundant scheme uses two extra *PE*s, while ABFT scheme uses one extra *PE* to tolerate a single fault. There is no degradation in the performance of the systolic array when ABFT scheme was used, while the throughput of the array decreases to one-third of its original throughput, if time redundancy is used for achieving fault tolerance. Another advantage of the ABFT scheme is that it does not require any reconfiguration once a fault is detected. The erroneous data elements are automatically corrected by the adjacent *PE*s. The time-redundancy scheme requires that a reconfiguration algorithm be run once an error is detected.

5. Theoretical aspects of ABFT

In the context of ABFT systems, two important problems have been identified. These are referred to as as (i) *the design* and (ii) *the analysis* problems. In the design problem, the goal is to construct a minimum collection of tests which can detect a specified number of errors/faults. In the analysis problem, the goal is to determine two important performance measures, namely the error-detectability (detection of erroneous data items) and the fault-detectability (detection of faulty processors), of a given ABFT system.

Graph theoretical and linear algebraic models (matrices) have been proposed to model ABFT [25–27]. In the graph theoretical model, an ABFT system is represented by a unidirectional tripartite graph [see Fig. 4(a)], called its *PDC graph*. $G=(P \cup D \cup C, E)$, where $P = \{p_1, p_2,..., p_m\}$ is a set of processors, $D = \{d_1, d_2,..., d_n\}$ is a set of data items, and $C = \{c_1, c_2,..., c_l\}$ is a non-empty set of checks. An edge $(p_i, d_j) \in E$ indicates that the data item d_j can be affected by a fault in p_i. An edge $(d_i, c_j) \in E$ indicates that the data item d_i is included in check c_j. We assume that each data item is affected by at least one processor and is included in at least one check, and each check contains at least one data item.

Nair and Abraham [28] proposed a matrix-based model for ABFT systems. They broke up the PDC graph of the graph theoretical model into two bipartite graphs: the PD graph and the DC graph. The two graphs were represented as PD and DC matrices. The product of the two matrices is called the PC matrix wherein the *ij*th entry corresponds to the number of paths from processor p_i to check c_j in the PDC graph.

5.1. Design problems

Research efforts have been concentrated on evaluating the minimum number of checks that are necessary to achieve a required degree of error/fault detectability and locatablilty. Banerjee and Abraham [25] proved bounds on

- data and error set cardinality;
- number of checks for fault detection;
- number of checks for fault location; and
- time and processor overhead.

The bounds on the number of checks for fault detection were derived subject to the checks being of (g, h) type and that there are at most s errors to be detected, where s is the cardinality of the largest error set generated by a pattern of t faults. The bounds for number of checks for fault location were derived for fault-locatable algorithms in which the results of a check on any two different error patterns are unique. The time and processor overheads are evaluated based on the assumption that each check is evaluated in the form of a Boolean expression evaluation and that each atomic operation requires 1 time unit.

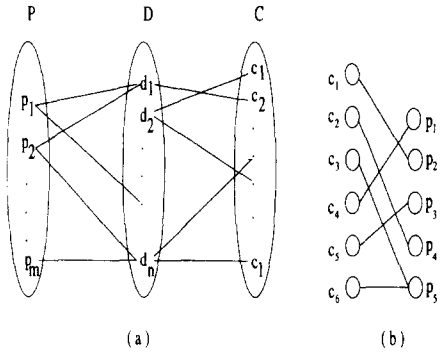


Fig. 4. (a) Tripartite graph representation of ABFT system—the PDC graph. (b) The check evalution graph.

Gu et al. [26] derived tighter bounds for the same model in which they proposed a divide-and-conquer strategy. A new method for construction of check sets which covers all the data elements in an f-fault tolerant system was designed in another work by the same authors [29]. However, they have only considered the error detection capabilities. The design of these check sets for a desired error correction capability is still a topic of research. Previous work on deterministic error detection had assumed a bound on the size of a check, whereas in Ref. [29] the size of a check is not bounded a priori, but depends on the size of the matrices involved.

Blough and Pelc [30] have used a probabilistic approach to model processor faults. They determined the probability with which errors can be detected and located in an ABFT system using graph theoretical methods. The number of checks that are required to guarantee error correction with high probability are also discussed in their work. They have, however, used transient faults model for analysis. When permanent faults are considered, all the data elements that are influenced by a faulty processor will be erroneous. It would be interesting to determine whether the number of checks required for diagnosis can be reduced in systems where both permanent and transient faults can occur.

5.2. Analysis problem

Although the graph theoretic and matrix models are natural and yield algorithms to determine the error-detectability and fault-detectability of a system, the resulting algorithms do not, in general, run in polynomial time. Srinivasan and Jha [31], however, proposed an efficient diagnosis algorithm which was based on the assumption that aliasing (discussed in Section 6) and hence check invalidation due to aliasing, is a rare phenomenon. Gu et al. [26] showed that the problem of determining the fault-detectability of a system is Co-NP-complete even when each data item is included in exactly two tests, but can be solved in linear time when each data item is included in only one test, but this implies that the check set cannot be minimal.

Yajnik and Jha [32] proposed a design method (an extended model) which considers the processors computing the checks to be a part of the ABFT system and guarantee concurrent error detection even in the presence of faults in these processors, unlike most methods presented earlier. Here, in addition to the PDC graph, a *check evaluation* (CE) bipartite graph, is presented (see Fig. 4b). The CE graphs consists of (i) *check_computing processor nodes* (p_i) and (ii) *check nodes* (c_j). There exists an edge between p_i and c_j, if c_j is evaluated on p_i. The CE graph is designed in such a way so as to maintain the fault detectability of the

system even in the presence of failures in the check_ computing processors. The PDC graph consists of (i) *info_processor nodes* (perform useful computations), (ii) *check_data nodes* (redundant part of the encoded output data), (iii) *check_nodes* (represent the checks), (iv) *code_ processor nodes* (perform computations on the encoded input to produce the check_data nodes), and (v) *real_data nodes* (results of useful computations). In Ref. [33], methods for analysing ABFT systems for fault detectability and localability under the extended model are discussed. Their methods were based on the assumption that there is no sharing of data nodes among the code_processor nodes, because this would make the fault localability impossible.

Finally, the application of these theoretical results to some actual algorithms, to show how fault diagnosis can be achieved in practice using the ABFT approach, remains untouched.

6. Limitations

In this section we highlight the main limitations of the ABFT approach.

1. In general ABFT can *significantly complicate algorithm* development. It can be applied to a handful of problems. ABFT schemes vary from algorithm to algorithm, and hence, designing an ABFT scheme for a problem would be equivalent to designing another algorithm (with redundant computation) for that problem. This might be one of the reasons for the limited use of ABFT as a means of achieving fault tolerance, compared to the traditional fault tolerant schemes.
2. *Limited PEs*. A faulty processing element (*PE*) may cause more than one element of the result to be erroneous if it is used repeatedly during a single matrix operation [6]. This can cause cascading of errors in the array of *PE*s with the result that the system may not actually be able to tolerate the errors.
3. *Masking of errors (aliasing of data elements)*. In a full checksum matrix, when an error is located in a row or a column which does not contain any other error, the error can be detected. However, when the errors can be connected to form a loop the errors may mask each other and cannot be detected. Aliasing occurs when an error in one or more data elements checked by some check is compensated by an error in another data element checked by that check. The output of this check is no longer valid and it must be invalidated. This reduces the fault-detectability of the system.
4. *Roundoff errors*. These errors are introduced in a system due to the limited number of bits available to represent a real number. This may result in *false alarms* where an error is diagnosed even though the system is fault free. To remedy this problem, a small difference η is allowed between the checksum and the actual output of the algorithm. This, however, leads to a trade-off based on the value of η. A small η surely degrades the system throughput since more retries are needed, but a large η may lead to many errors escaping detection.

7. Conclusions

We have discussed the concepts of ABFT techniques for adding fault tolerance to computer systems in a cost-effective manner. The effectiveness of ABFT techniques were demonstrated by some matrix operations. A dependency graph-based ABFT scheme in systolic arrays was also discussed, and the overheads in this scheme were compared to those in the traditional triple time redundancy. Although ABFT offers a promising solution for adding fault tolerance, in certain problems, in terms of low cost, it has certain limitations which severely limits its practical applicability. It has been applied only to a handful of problems and it is not a general fault-tolerant scheme. The theoretical basis for ABFT was presented. These theoretical aspects have yet to be applied to certain practical problems. For example, the problem of determining the minimum number of checks required to perform matrix operations in a fault-tolerant manner is still to be investigated. Furthermore, the application of ABFT to certain other multiprocessor architectures remains unexplored.

Acknowledgements

The authors would like to thank Dr Siva Ram Murthy (I.I.T., Maddras) for useful discussion.

References

[1] S.H. Hosseini, On fault tolerant structure, distributed fault diagnosis, reconfiguration, and recovery of the array processors, IEEE Transactions of Computing 38 (7) (1989) 932–942.
[2] M.O. Esonu, A.J. Al-Kalili, S. Hariri, D. Al-Kalili, Fault tolerant design methodology for systolic array architectures, IEE Proceedings E 141 (1) (1994) 17–28.
[3] S.-Y. Kuo, S.-C. Liang, Concurrent error detection and correction in real time systolic arrays, IEEE Transactions of Computing 41 (12) (1992) 1615–1620.
[4] A. Majumdar, C.S. Raghavendra, M.A. Breuer, Fault tolerance in linear systolic arrays using time redundancy, IEEE Transations of Computing 39 (12) (1990) 269–276.
[5] A.D. Singh, Interstitial redundancy: an area efficient fault tolerance scheme for large area VLSI processor arrays, IEEE Transations of Computing 37 (11) (1988) 1398–1410.
[6] K.H. Huang, J.A. Abraham, Algorithm-based fault tolerance for matrix operations, IEEE Transations of Computing 33 (1984) 518–528.
[7] P.Banerjee, J.A.Abraham, Fault secure algorithms for multiple processor systems, Proceedings of the 11th International Symposium on Computing Architecture, Ann Arbor, MI, June, 1984, pp. 279–287.
[8] S.-J. Wang, N.K. Jha, Algorithm-based fault tolerance for FFT networks, IEEE Transations of Computing 43 (7) (1994) 849–854.

[9] C.-G. Oh, H.-Y. Youn, V.K. Raj, An efficient algorithm-based concurrent error detection for FFT networks, IEEE Transations of Computing 44 (9) (1995) 1.

[10] G.L. Feng, T.R.N. Rao, M.S. Kohulru, Error correcting codes over Z_{2^m} for algorithm-based fault tolerance, IEEE Transations of Computing 43 (3) (1994) 370–374.

[11] J.Y. Jou, J.A. Abraham, Fault tolerant matrix operations on multiple systems using weighted checksums, Real Time Signal Processing, Proceedings of the SPIE 495 (1984) 94–101.

[12] F. Luk, Algorithm-based fault tolerance for parallel matrix equation solvers, SPIE, Real Time Signal Processing 564 (1985) 49–53.

[13] Y.-M. Yeh, T.-Y. Feng, Algorithm-based fault tolerance for matrix inversion with maximum pivoting, J. Parallel and Distributed Computing 14 (1992) 373–389.

[14] J. Rexford, N.K. Jha, Partitioned encoding schemes for algorithm-based fault tolerance in massively parallel systems, IEEE Transations of Parallel and Distributed systems 5 (6) (1994) 649–653.

[15] J.I. Khan, W. Lin, D.Y.Y. Yun, Adaptive algorithm based-fault tolerance for parallel computing in linear systems, International Conference on Parallel Processing, 1994.

[16] P. Banerjee, J.T. Rameh, C. Stunkel, V.S. Nair, K. Roy, V. Balasubramanian, J.A. Abraham, Algorithm-based fault tolerance on a hypercube multiprocessor, IEEE Transactions on Computers 39 (9) (1990) 1132–1145.

[17] Y.H. Choi, M. Malek, A fault tolerant FFT processor, IEEE Transations of Computing C37 (1988) 617–621.

[18] J.Y. Jou, J.A. Abraham, Fault tolerant FFT networks, IEEE Transations of Computing C37 (1988) 548–561.

[19] A.L.N. Reddy, P. Banerjee, Algorithm-based fault detection for signal processing applications, IEEE Transations of Computing 39 (1990) 1304–1308.

[20] D.L.Tao, C.R.P.Hartmann, Y.S.Chen, A novel concurrent error detection scheme for FFT networks, Proceedings of the International Symposium on Fault-Tolerant Computing, June, 1990, pp. 114–121.

[21] S.Y. Kung, VLSI Array Processors, Prentice Hall, Englewood Cliffs, NJ, 1988.

[22] H.T. Kung, Why systolic architectures?, IEEE Computing Magazine January (1982) 37–46.

[23] B. Vinnakota, N.K. Jha, Synthesis of algorithm-based fault tolerance systems from dependence graphs, IEEE Transations of Parallel and Distributed Systems 4 (8) (1993) 864–874.

[24] M. Vijay, C. Siva Ram Murthy, Fault-tolerant systolic evaluation of polynomials and exponentials of polynomids for equispaced arguments, IEEE Transactions on Circuits and Systems 44 (3) (1997) 246–250.

[25] P. Banerjee, J.A. Abraham, Bounds on algorithm-based fault tolerance in multiple processor systems, IEEE Transations of Computing C35 (4) (1986) 296–306.

[26] D. Gu, D.J. Rosenkrantz, S.S. Ravi, Determining performance measures of algorithm-based fault tolerant systems, Journal of Parallel and Distributed Computing 18 (1993) 56–70.

[27] C.J. Anfinson, F.T. Luk, A linear algebraic model of algorithm-based fault tolerance, IEEE Transactions on Computers 37 (12) (1988) 1599–1604.

[28] V.S.S. Nair, J.A. Abraham, A model for the analysis of fault tolerant signal processing architectures, Proceedings of the International Technology Symposium SPIE, August, 1988, pp. 246–257.

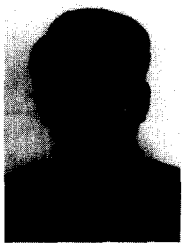
[29] D. Gu, D.J. Rosenkrantz, S.S. Ravi, Construction of check sets for algorithm-based fault tolerance, IEEE Transactions on Computers 43 (6) (1994) 641–650.

[30] D.M. Blough, A. Pelc, Almost certain fault diagnosis through algorithm-based fault tolerance, IEEE Transactions on Parallel and Distributed Systems 5 (5) (1994) 532–539.

[31] S. Sriniyasan, N. K. Jha, Efficient diagnosis in algorithm-based fault tolerant multiprocessor systems, IEEE International Conference on Computer Design, Boston, MA, October, 1993.

[32] S. Yajnik, N. K. Jha, Analysis and randomized design of algorithm-based fault tolerant microprocessor systems under the extended graph-theoretic model, ISCA International Conference on Parallel and Distibuted Systems, Louisville, October, 1993.

[33] S. Yajnik, N. K. Jha, Design of algorithm-based fault tolerant systems with in-system checks, International Conference on Parallel Processing, St Charles, IL, August, 1993.



Ravi Mittal obtained B.Engg. (Electronics and Telecommunication Engineering) in 1983 from the University of Jabalpur, M. Tech. (Computer Science and Technology) in 1985 from the University of Roorkee, and Ph.D. in 1991 from the Indian Institute of Technology, Delhi, India. Since 1990, he is with the Department of Computer Science and Engineering, where he is presently Assistant Professor. His research interests include high speed networking, Mobile computing, interconnection networks, fault-tolerant and real-time systems.