Algorithm-Based Fault Tolerance for Matrix Operations

KUANG-HUA HUANG, MEMBER, IEEE, AND JACOB A. ABRAHAM

Abstract — **The rapid progress in VLSI technology has reduced the cost of hardware, allowing multiple copies of low-cost processors to provide a large amount of computational capability for a small cost. In addition to achieving high performance, high reliability is also important to ensure that the results of long computations are valid. This paper proposes a novel system-level method of achieving high reliability, called *algorithm-based fault tolerance*. The technique encodes data at a high level, and algorithms are designed to operate on encoded data and produce encoded output data. The computation tasks within an algorithm are appropriately distributed among multiple computation units for fault tolerance. The technique is applied to matrix computations which form the heart of many computation-intensive tasks. Algorithm-based fault tolerance schemes are proposed to detect and correct errors when matrix operations such as addition, multiplication, scalar product, *LU*-decomposition, and transposition are performed using multiple processor systems. The method proposed can detect and correct any failure within a single processor in a multiple processor system. The number of processors needed to just detect errors in matrix multiplication is also studied.**

Index Terms — **Algorithm-based fault tolerance, checksum matrix, error correction, error detection, matrix operations, multiple processor systems, processor arrays, systolic arrays, transient errors.**

I. INTRODUCTION

MANY scientific research areas require a large amount of high computation power to solve their problems. The rapid progress in VLSI technology has reduced the cost of hardware, and a cost-effective means of achieving high system performance is to use multiple copies of identical processors. Matrix operations are performed in the inner loops of computation-intensive tasks such as signal and image processing, weather prediction, and finite element analysis. These operations often constitute the performance bottleneck for such tasks. Multiple processor systems such as massively parallel processor (MPP) [1] and systolic arrays [2] have been proposed to solve this problem.

In addition to achieving high system performance, it is important that reliable results be obtained from such systems. High reliability could be achieved by the use of existing fault-tolerant systems. Fault tolerance can be obtained either by masking the errors caused by physical failures or by detecting them, isolating the faulty unit, and reconfiguring the system around the faulty unit. Fault masking is usually done by replicating the hardware and voting on the outputs of the replicated modules, but this is extremely costly in terms of hardware. An alternative is to detect errors by testing or by coding techniques. Unfortunately, the increase in transient errors caused by decreasing geometries means that off-line testing alone cannot be used to detect erroneous modules. Techniques are therefore needed to detect errors concurrently with normal operation.

Existing fault-tolerant techniques which are applicable to transient errors can be divided into two categories: error masking, and concurrent error detection followed by reconfiguration. The error masking techniques, which tolerate failures and provide continuous system operation, include triple modular redundancy (TMR) [3] and quadded logic [4]; these require at least a factor of 2 or 3 in additional hardware redundancy to tolerate single module failures. Concurrent error detection techniques, which are designed to signal errors but not mask them, include totally self-checking (TSC) circuits [5], alternating logic [6], recomputing with shifted operands (RESO) [7], and watchdog processors [8]. Examples using TSC techniques require 73 percent hardware redundancy in a processor for detecting single bit errors and 94 percent for detecting unidirectional errors [9]. Alternating logic requires 100 percent time redundancy plus an average value of 85 percent redundancy in hardware to detect a single failure [6]. RESO is an effective technique for applications where the CPU is not busy all the time; however, for computation-intensive tasks, it uses 100 percent time redundancy for recomputation and comparison of results to detect errors. All of these detection techniques require backup hardware for fault tolerance. Also, many of these techniques are based on the single stuck-at fault model, which cannot cover all the physical failures in VLSI chips [10].

This paper proposes a new technique called *algorithm-based fault tolerance* which can be used to detect and correct errors caused by permanent or transient failures in the hardware. This technique is not as generally applicable as some of the classical techniques such as TMR (which can be applied to any module); however, in the specific cases where it applies, fault tolerance can be achieved with a surprisingly low overhead. The technique is applied specifically to matrix operations in this paper.

In Section II, a module level fault model applicable to VLSI is described. Section III describes the general technique, and the design of algorithms to achieve fault tolerance in matrix operations is described in Section IV. Modi-

Manuscript received August 12, 1983; revised December 12, 1983 and January 20, 1984. This research was supported by the Naval Electronics Systems Command under VHSIC Contract N00039-80-C-0556. A part of this paper was presented at the 12th International Symposium on Fault-Tolerant Computing, Santa Monica, CA, June 1982.

K.-H. Huang was with the Coordinated Science Laboratory, University of Illinois, Urbana, IL 61801. He is now with the Engineering Research Center, AT&T Technologies, Inc., Princeton, NJ 08540.

J. A. Abraham is with the Coordinated Science Laboratory, University of Illinois, Urbana, IL 61801.

0018-9340/84/0600-0518$01.00

fications of existing processor arrays to exploit the algorithms are given in Section V, and the overhead in hardware and time for fault tolerance determined in each case. The number of processors required to just detect errors in matrix multiplication is also discussed.

II. A Module Level Fault Model

As the geometric features of integrated circuits become smaller, any physical defect which affects a given area on a chip will affect a greater amount of the circuitry and will cause a larger block of logic to become faulty. Thus, the commonly used gate-level single stuck-at fault models are not satisfactory. In this paper we will use a very general fault model, which allows a module (such as a processor or computation unit in a multiple processor system) to produce, under failure, any erroneous output values which are clearly logic 0 or 1 when they are interpreted by another module. We will assume that at most one module is faulty within a given period of time which will be reasonably short compared to the mean time between failures; this assumption is reasonable if periodic testing is performed to flush out the latent failures.

The processors communicate with each other and with memory, and we must account for failures in the communication lines. Fortunately, effective error correcting schemes such as Hamming codes [11] and alternate retry [12] exist for communication lines and memories. The input/output latch registers of a processor can be counted as part of the communication circuitry since a fault in the latch registers affects the data transfer but not the computation itself. In this paper, we will therefore focus on the fault tolerance of the processors.

III. Algorithm-Based Fault Tolerance

This section describes the basic ideas behind the algorithm-based fault tolerance technique.

Hardware failures within the modules of a system can be tolerated by the use of redundant modules which can perform the same computation (TMR, for example). Such a technique is quite general, and can be used to protect any computational module from failures. The price which has to be paid for this generality is, unfortunately, the high cost of the overall fault-tolerant system. This paper proposes a novel approach which achieves fault tolerance with extremely low cost by tailoring the fault tolerance scheme to the algorithm to be performed. As will be seen later, the technique is especially useful for high-performance systems where multiple processors are used for high throughput. Fault tolerance is particularly necessary in such systems because of the large amount of hardware and the high computation rate, and only a few additional processors are needed to tolerate failures in our scheme.

Algorithm-based fault tolerance is distinguished by three characteristics: the encoding of the data used by the algorithm, the redesign of the algorithm to operate on the encoded data, and the distribution of the computation steps in the algorithm among computation units. These will be described in more detail below.

Conventional data encoding is usually done at the word level in order to protect against errors which affect bits in a word. Since a faulty module could affect all the bits of a word it is operating on, we need to encode data at a higher level. This can be done by considering the set of input data to the algorithm and encoding this set; in the matrix schemes to follow, we will use row and column checksums as the encoding, for example.

The original algorithm must then be redesigned to operate on these encoded data and produce encoded output data. The information portion of the encoded data must be easy to recover, in order that the scheme be practical. The modified algorithm could, of course, take more time to operate on the encoded data when compared to the original algorithm, and this time overhead must not be excessive.

Finally, the computation tasks within the algorithm must be appropriately distributed among multiple computation units, so that failure of any unit affects only a portion of the data. The redundancy in the encoding would enable the correct data to be recovered. The error detection and correction schemes must be designed so that a faulty module (which caused erroneous data in the first place) will not mask the error during the detection or correction steps.

The remainder of the paper develops an algorithm-based fault tolerance scheme for matrix operations when they are performed in different types of multiple processor systems.

IV. A Matrix Encoding Scheme and the Computations Preserving the Encoding

The encoding for matrices is obtained from the two-dimensional product code [13], but the encoding is done at the word (integers or floating point numbers) level rather than at the bit level. The encoded matrices are called *checksum matrices* in this paper. Algorithms are designed to operate on these encoded matrices to produce encoded output matrices. Checksum matrix schemes have been used to avoid inaccuracies due to roundoff errors [21], [22]. Since there is redundancy in the encoding, they have also been used as error checks in desk calculations [21]. This paper will discuss systematic methods of using the checksum matrices to detect and correct errors in computations with multiple processors.

The checksum matrix technique is different from the checksum codes used in arithmetic units or in communication paths for error detection; these codes are designed to detect errors at the bit level in words or characters. Here, the encoding is at the vector or matrix level. For example, the optimal rectangular code (ORC) presented in [14] is a binary product code. The ORC is designed to correct errors in tapes (data storage) and the checksum matrices are designed to correct errors in computations. The results here will be used later to design fault-tolerant multiple processor structures for matrix operations.

The row, column, and full checksum matrices for an n-by-m matrix A (denoted by $A_{n,m}$) with elements $a_{i,j}$ are defined as follows. (The elements $a_{i,j}$ could be either integers or floating point numbers.)

Definition 4.1: The *column checksum matrix* A_c of the matrix A is an $(n + 1)$-by-m matrix, which consists of the matrix A in the first n rows and a *column summation vector*

in the $(n + 1)$st row; the elements of the summation vector are generated as

$$a_{n+1,j} = \sum_{i=1}^{n} a_{i,j} \qquad \text{for } 1 \le j \le m. \tag{1}$$

Using the notation in [21], $A_c = \left|\frac{A}{e^T A}\right|$, where e^T is a 1-by-n vector [1 1 1 1 $\cdots$ 1] and the vector $e^T A$ is the column summation vector.

Definition 4.2: The *row checksum matrix* A_r of the matrix A is an n-by-$(m + 1)$ matrix which consists of the matrix A in the first m columns and a *row summation vector* in the $(m + 1)$st column; the elements of the summation vector are generated as

$$a_{i,m+1} = \sum_{j=1}^{m} a_{i,j} \qquad \text{for } 1 \le i \le n. \tag{2}$$

$A_r = |A\,|\,Ae|$, where Ae is the row summation vector.

Definition 4.3: The *full checksum matrix* A_f of the matrix A is an $(n + 1)$-by-$(m + 1)$ matrix, which is the column checksum matrix of the row checksum matrix A_r.

Definition 4.4: Each row or column in the full checksum matrix is called a *checksum encoded vector* and is denoted by *CSEV*.

From the definitions, we can see that each checksum matrix has its separate information matrix (A) and summation vectors. To apply the checksum technique, each matrix is stored in its full checksum matrix format and is manipulated in the full, row, or column checksum matrix format depending on the matrix operations. Five matrix operations exist which preserve the checksum property; they are given in the following theorems. We use the symbol "*" for both matrix and scalar multiplication; it is clear from the context which operation is intended.

Theorem 4.1: (i) The result of a column checksum matrix (A_c) multiplied by a row checksum matrix (B_r) is a full checksum matrix (C_f). (ii) The corresponding information matrices A, B, and C have the following relation:

$$A * B = C.$$

Proof:

$$A_c * B_r = \left|\frac{A}{e^T A}\right| * |B\,|\,Be| = \left|\frac{AB}{e^T AB}\,\middle|\,\frac{ABe}{e^T ABe}\right| = C_f. \qquad \square$$

Fig. 1 depicts the checksum matrix multiplication.

LU decomposition of a matrix is a time-consuming part of the procedure used to solve large linear equations

$$C * x = b$$

where C is an $n * n$ matrix, b is a given $n * 1$ vector, and x is an unknown $n * 1$ vector.

If the equation $C * x = b$ can be solved by Gaussian elimination without pivoting, then the matrix C, with elements $c_{i,j}$, can be decomposed into the product of a lower triangular matrix with an upper triangular matrix

$$C = L * U$$

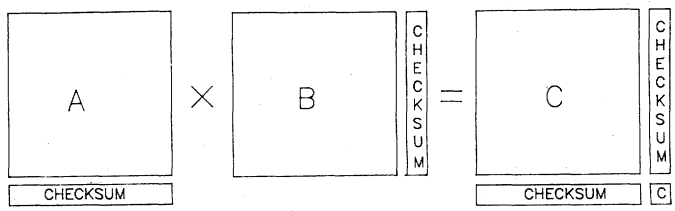

Fig. 1. A checksum matrix multiplication.

where $U = (u_{i,k})$ and $L = (l_{k,j})$ are evaluated [2] as follows:

$$^1c_{i,j} = c_{i,j}$$

$$^{k+1}c_{i,j} = {}^kc_{i,j} + l_{i,k}(-u_{k,j})$$

$$l_{i,k} = \begin{cases} 0 & \text{when } i < k, \\ 1 & \text{when } i = k, \\ {}^kc_{i,k} * (1/u_{k,k}) & \text{when } i > k, \end{cases}$$

$$u_{k,j} = \begin{cases} 0 & \text{when } k > j, \\ {}^kc_{k,j} & \text{when } k \le j. \end{cases}$$

If the pivoting is required in order for the procedure to work, then C can be factored into L and U; but, in general, they are not triangular matrices [15].

From [21, p. 265] we get the following theorem.

Theorem 4.2: When the information matrix C is *LU* decomposable, the full checksum matrix of C, C_f, can be decomposed into a column checksum lower matrix and a row checksum upper matrix.

Proof: Let the decomposition of C be $C = LU$,

$$C_f = \left|\frac{C}{e^T C}\right|\frac{Ce}{e^T Ce}\right|$$

can be decomposed as $L_l U_u$ where

$$L_l = \left|\frac{L}{e^T L}\right| \quad \text{and} \quad U_u = |U\,|\,Ue|. \qquad \square$$

Theorem 4.3: (i) The result of the addition of two full checksum matrices (A_f and B_f) is a full checksum matrix (C_f). (ii) Their corresponding information matrices have the relation

$$A + B = C. \qquad \square$$

Corollary 4.1: The result of the addition of two CSEV's is a CSEV. □

Theorem 4.4: The product of a scalar and a full checksum matrix is a full checksum matrix. □

Corollary 4.2: The product of a scalar and a CSEV is a CSEV. □

Theorem 4.5: The transpose of a full checksum matrix is a full checksum matrix. □

Matrix addition, multiplication, scalar product, *LU* decomposition, and transpose thus preserve the checksum property.

A. Effect on the Word Length when Using the Checksum Technique

Since the checksum elements are the sum of several matrix elements, we must consider the possible problems with word

lengths. When 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 4 in the exponent for base 16 systems and 16 for base 2 systems.

When processing the matrix with integer elements, the same word length r can be used to store the elements of summation vectors if residue arithmetic [16] is applied to the computations on the summation vectors with a modulus $M = 2^r$. Equations (1) and (2) in Definitions 4.1 and 4.2 must be modified, however, to be

$$a_{n+1,j} = \sum_{i=1}^{n} a_{i,j} \bmod M \qquad \text{for } 1 \le j \le m. \tag{3}$$

$$a_{i,m+1} = \sum_{j=1}^{m} a_{i,j} \bmod M \qquad \text{for } 1 \le i \le n. \tag{4}$$

A summation vector is obtained simply by performing residue arithmetic on the original matrix elements modulo 2^r.

From the properties of residue arithmetic [16],

$$(x + y) \bmod q = (x \bmod q + y \bmod q) \bmod q$$

and

$$(x * y) \bmod q = (x \bmod q * y \bmod q) \bmod q.$$

Theorems 4.1–4.5 are also true when the summation vectors of a checksum matrix are defined by (3) and (4).

B. Characteristics of Checksum Matrices

The definitions of matrix distance and vector distance are introduced in this section; they will be used to verify the fault-tolerance possibilities of checksum matrices.

Definition 4.5: The *matrix (vector) distance* between two matrices (vectors) is the number of elements in which they differ.

Definition 4.6: The *weight* of a matrix is the number of nonzero elements.

Let $S_{p,q}$ be the set of all (unique) p by q full checksum matrices.

Definition 4.7: The *minimum matrix distance* of a set of full checksum matrices is the minimum of the matrix distances between all possible pairs of full checksum matrices in the set.

Theorem 4.6: The minimum matrix distance of $S_{p,q}$ is 4.

Proof: A full checksum matrix is a product code [13] since each row and column of a full checksum matrix is a distance 2 vector. From Elias' Theorem [17], we know that the minimum weight of a nonzero full checksum matrix is 4. For any two matrices A and B belonging to $S_{p,q}$ and $A \neq B$, let matrix C be

$$C = A - B = A + (-1)B.$$

From Theorems 4.3 and 4.4, we know that the matrix C belongs to $S_{p,q}$. So, the weight of matrix C is equal to or larger than 4. Thus, the minimum matrix distance of $S_{p,q}$ is 4. [ ]

Therefore, a single erroneous element can be corrected in a full checksum matrix.

C. Error Detection, Location, and Correction

When we use as many processors as necessary to perform matrix operations, a processor only performs computations for one or a few elements of the output matrix; thus, only a limited number of elements could be incorrect. For the case of only one erroneous element in a full checksum matrix, exactly one row and one column will have an incorrect checksum. The intersection of this row and column locates the erroneous matrix element. A procedure used to detect, locate, and correct a single erroneous element in a full checksum matrix is listed in the following.

1) Error Detection:

a) Compute the sum of information elements in each row and column.

b) Compare each computed sum to the corresponding checksum in the summation vector. (Note: since there may be roundoff errors for floating point operations, a small tolerance [18] should be allowed for in the comparison.)

c) An *inconsistent row* or *column* is detected by an inequality in b).

2) Error Location:

a) An error is located at the intersection of the inconsistent row and inconsistent column.

3) Error Correction:

a) The erroneous element can be corrected (i) by adding the difference of the computed sum of the row or column data elements and the checksum to the erroneous element in the information part, (ii) or by replacing the checksum by the computed sum of the information elements in the summation vector, in the case where the checksum is incorrect.

A large roundoff error in an element of the output matrix can be detected and treated in the same manner as a processor with a transient fault. When a large roundoff error occurs in the checking procedure (but not in the computation of the elements), only one inconsistent row or column is detected, and this can be ignored. The procedure will detect the case where more than one element has a large roundoff error; the probability of detection will be better than that for a uniprocessor (discussed in Section V-D).

Clearly, "false alarms" are possible where roundoff errors may cause the checks to indicate that a processor is faulty. Prevention of such alarms requires a detailed analysis of the numerical aspects of the problem and is beyond the scope of this paper. We believe that it is better, in many cases, to obtain such an alarm for a large roundoff error, rather than to have an inaccurate result.

V. Application of the Checksum Matrix Techniques to Multiple Processor Systems

Checksum matrix operations produce code outputs which provide some degree of error-detecting and correcting capability, as discussed in Section IV. However, a faulty module may cause more than one element of the result to be erroneous if it is used repeatedly during a single matrix operation. Therefore, the process of redesigning the algorithm and distributing the computation steps among multiple processing units should be considered carefully to prevent masking of errors; examples are given in this section.

In Section V-A, the checksum technique is applied to mesh-connected processor arrays for dense matrix multiplication, and to systolic processor arrays for band matrix multiplication. We will investigate the fault tolerance and required redundancy for each architecture. In Sections V-B and V-C, the checksum technique is applied to processor arrays for matrix LU-decomposition and matrix inversion. Section V-D will discuss the error coverage of the checksum technique used with uniprocessors.

A. Matrix Multiplication

In this section, the checksum matrix scheme is used to detect and correct errors when matrix multiplications are performed using mesh-connected processor arrays and systolic arrays. These architectures are investigated and compared from a fault-tolerance viewpoint; it is shown that only small redundancy ratios—$O(1/n)$ for hardware and $O(\log_2(n)/n)$ for time—are required for processor array systems to achieve fault-tolerant matrix operations. The *redundancy ratio* is defined as the ratio of the hardware or time overhead required by the fault-tolerance technique to the hardware or time complexity of the original system without fault tolerance. We also investigate the minimum number of processors required to produce a checksum matrix in a matrix multiplication in which errors caused by a faulty processor can be detected.

1) Mesh-Connected Processor Arrays: Fig. 2 shows a small mesh-connected processor array with row/column broadcast capability [19]. (A row/column broadcast processor array consists of a common data bus for each row or column of processors, and data can be broadcast to all processors in the row or column at the same time.) In this example, the input data streams for the multiplication of two 4-by-4 matrices with summation vectors are also indicated.

The operation of the array is as follows. For the matrix elements $a_{i,j}$ on the left-hand side of the array, the top four rows consist of the elements of the information part of matrix A_c and the fifth row is the summation vector; the elements $a_{i,j}$ are broadcast from the left boundary of the array to processors in the ith row at time j. For the matrix elements $b_{j,k}$ on the top of the array, the leftmost four columns comprise the elements of the information part of matrix B_r and the fifth column is the summation vector; the elements $b_{j,k}$ are broadcast to processors in the kth column at time j. At time j, the processor $P_{i,k}$, which is located on the intersection of the ith row and kth column of the array, performs the product of $a_{i,j}$ and $b_{j,k}$ and accumulates the product in a register. Thus, after n time steps (4, in the example), each processor calculates an element of the matrix C_f and any single error can be located and corrected, as mentioned in Section IV.

The redundancy required for performing the checksum matrix multiplication in an $(n + 1)$-by-$(n + 1)$ mesh-connected array is discussed here; the $(n + 1)$th row and column processors are used to calculate the summation vectors. These $2n + 1$ processors are the hardware overhead required by the checksum technique. After the result matrix C_f is obtained, we can use the first n processors of the $(i + 1)$th row of the processor array to calculate the sum of the ith row of the matrix C in $\log_2(n)$[1] time units of scalar addition, where $1 \le i \le n$. This will prevent a faulty processor in the ith row from producing a sum which will mask the previous error. The $(n + 1)$th row of the matrix C_f can be similarly summed by the first row of the processor array. An additional time unit is needed to compare the computed sum to the corresponding element in the summation vector to detect any inconsistent row. The same amount of time—$\log_2(n)$ units—is required to check the consistency of the columns of the matrix C_f. Table I shows the complexity of a nonchecksum matrix multiplication performed in a mesh-connected processor array and the redundancy required by the multiplication with the checksum technique.

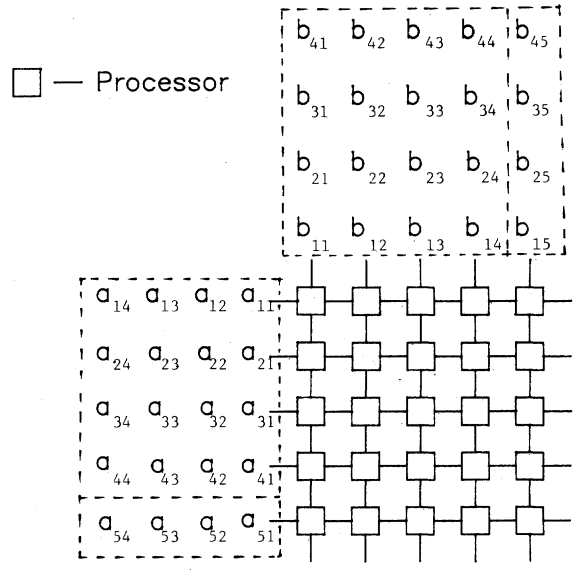

Fig. 2. Multiplication of two dense checksum matrices in a mesh-connected processor array.

Multiplication of Large Matrices with a Limited Number of Processors

So far, we have only considered the case where each processor performs computations for a single output element. In reality, the number of processors in a system is limited; the multiplication of larger matrices with a limited number of processors is discussed in this subsection.

A matrix can be partitioned into several submatrices and each submatrix can be encoded to be a checksum matrix to fit the size of a given processor array. The multiplication of partitioned checksum matrices shown in Fig. 3 will produce a matrix with multiple summation vectors as shown in Fig. 3; each small strip is a summation vector of the submatrix on its top or left. In each checksum submatrix there is, at most, one erroneous element; thus, the error can be detected and corrected. Such a method requires more than one row and more than one column of summation vectors.

Fig. 4 shows how a p-by-p processor array can perform operations on an n-by-n ($n = \lfloor (p - 1)/2 \rfloor * p$) checksum

[1]Assuming the data transfer time is negligible. If the data transfer time is not negligible, either $O(n)$ checkers or the boundary processors can be used to calculate the checksums without increasing the order of the redundancy ratio.

TABLE I
THE COMPLEXITY OF A MATRIX MULTIPLICATION PERFORMED IN A MESH-CONNECTED PROCESSOR ARRAY AND THE REDUNDANCY REQUIRED BY THE CHECKSUM TECHNIQUE

	The complexity of the matrix multiplication without fault-tolerance techniques	The redundancy required with the checksum technique	Redundancy ratio
processor	n^2	$2n + 1$	$2/n$
time	n	$2 * k * \log_2(n)$	$2 * k * \log_2(n)/n$

k is the ratio of execution time of an addition to that of a multiplication performed in a processor array.

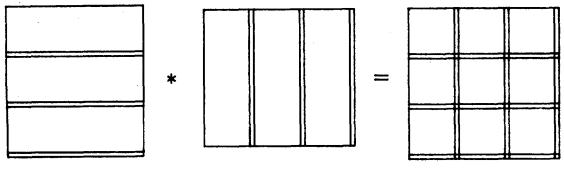

Fig. 3. Partitioned checksum matrices.

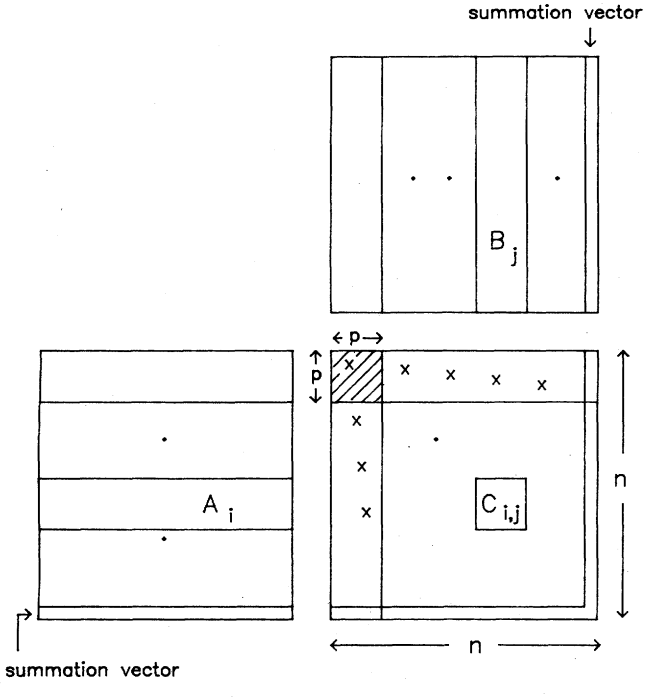

Fig. 4. An operation on a checksum matrix performed in an array of smaller dimension.

matrix with the errors caused by a faulty processor being corrected. The operation is described as follows.

1) Partition a column checksum matrix A_c into p-by-n submatrices A_i and partition a row checksum matrix B_r into n-by-p submatrices B_j.

2) Rotate[2] A_i down $(j - 1)$ rows and B_j to the right $(i - 1)$ columns and feed them into the array to compute the elements in the submatrix $C_{i,j}$.

3) Repeat step 2) until all of the submatrices are obtained.

During the computation, a faulty processor in the array will cause an element in each submatrix $C_{i,j}$ to be unreliable. Then there are, at most, $\lfloor (p - 1)/2 \rfloor$ erroneous elements in the first p rows of the output matrix C_f since there are $\lfloor (p - 1)/2 \rfloor$ p-by-p submatrices. The rotations on the input submatrices A_j in Step 2) cause the erroneous elements to be located in $\lfloor (p - 1)/2 \rfloor$ consecutive[3] rows in the first p rows of C_f, and each row contains, at most, one erroneous element. Therefore, the errors can be detected from the checksum property of each row since there are at least $\lfloor (p + 1)/2 \rfloor$ consecutive rows in the first p rows which are consistent. Similarly, at least $\lfloor (p + 1)/2 \rfloor$ consecutive columns in the first p columns of C_f are consistent, and each inconsistent column in the first p columns contains, at most, one erroneous element. Thus, the faulty processor can be located at the intersection of the first inconsistent row and the first inconsistent column after the $\lfloor (p + 1)/2 \rfloor$ consecutive consistent rows and columns in submatrix $C_{1,1}$.

From the operations described above, the processor $P(x, y)$, located at coordinates (x, y) in the array, performs the computation for the element $c(x + i - 1, y + j - 1)$ in submatrix $C_{i,j}$; thus, once the faulty processor is located, the set of unreliable elements is located. Furthermore, the rotations in Step 2) also cause each row of C_f to contain, at most, one erroneous element similar to the first p rows of matrix C_f. Therefore, all the errors caused by a faulty processor can be corrected.

2) Matrix Multiplication Systolic Arrays: Fig. 5 shows the multiplication of two checksum band matrices A_c and B_r. The matrix A_c consists of a band matrix A and a column summation vector; matrix B_r consists of a band matrix B and a row summation vector. The widths of band matrices A and B are $W1$ and $W2$, respectively. The output matrix C_f is a band matrix C with its column and row summation vectors. Fig. 6 shows the array structure, the data streams, and the redundant hardware required by the checksum technique for the matrix multiplication in Fig. 5. The systolic algorithm used in this section is the new efficient algorithm presented in [20].

Based on the algorithm, matrix A is fed into the systolic array from the left top boundary, B from the right top, and C from both sides of the top. All elements in the bands of matrices A, B, and C move synchronously through the array in three directions. Each $c_{i,k}$ is initialized to zero as it enters the array and accumulates all its partial product terms before it leaves the array through the bottom boundary.

Fig. 6 shows a large module $G1$ which consists of $W1$ inner product processors, $W1 - 1$ full word adders, and $\min(W1, W2) + 1$ buffers. The module $G1$ receives the data streams $a_{i,j}$ at the same time as the processor array. Thus, no extra memory requests are required. The module $G1$ also receives the row summation vector of matrix B_r via the rightmost processor, and it performs computations with these two

[2]Rotating a matrix down by one row moves the ith row to the $(i + 1)$st row and the last row to the first row; similarly, rotating a matrix to the right by one column moves the ith column to the $(i + 1)$st column and the last column to the first column.

[3]Assume the last row of a submatrix is adjacent to the first row, and the last column is adjacent to the first column.

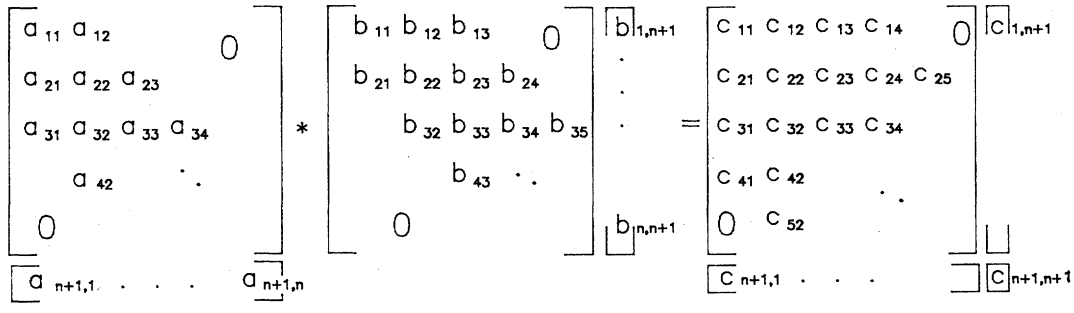

Fig. 5. The matrix multiplication of two checksum band matrices.

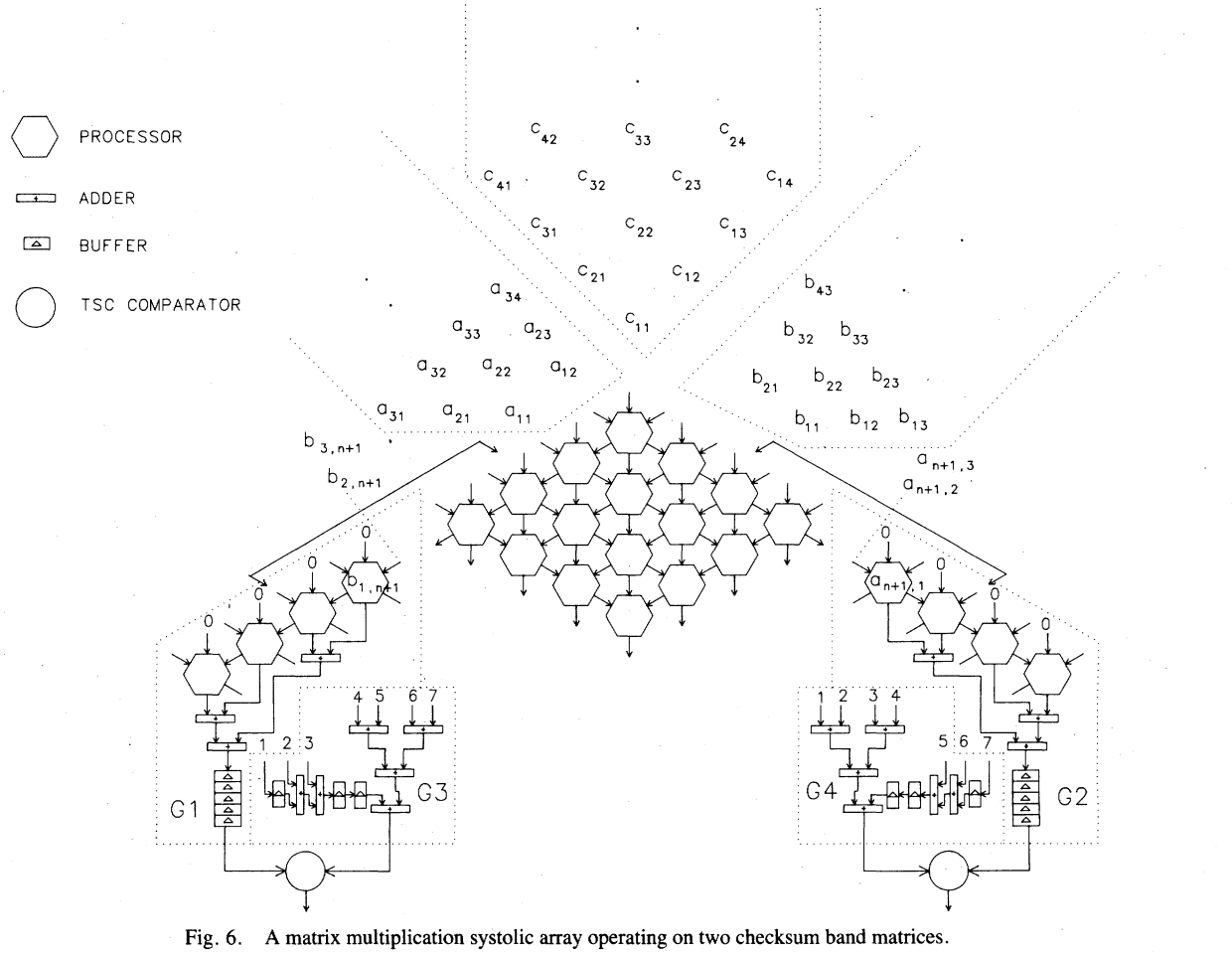

Fig. 6. A matrix multiplication systolic array operating on two checksum band matrices.

input streams to produce the elements of the row summation vector of the matrix C_f. Similarly, module $G2$, consisting of $W2$ inner product processors, $W2 - 1$ full word adders, and $\min(W1, W2) + 1$ buffers, produces the column summation vector of the matrix C_f.

Two more modules, $G3$ and $G4$, are also shown in Fig. 6. Each module consists of $W1 + W2 - 1$ full word adders, $\log_2(W2)$ buffers for the module $G3$, and $\log_2(W1)$ buffers for the module $G4$. Each input bus of a module is connected to a corresponding output bus (in the bottom) of the systolic array as shown in the figure. Thus, the modules $G3$ and $G4$ receive the matrix elements $c_{i,k}$ from the systolic array to compute the sum for each column and row of matrix C, respectively. The outputs of module $G3$ are compared to the outputs of module $G1$ by a TSC comparator to judge whether the columns of the matrix C_f satisfy the checksum property. The outputs of modules $G2$ and $G4$ are similarly compared to check the rows of the matrix C_f. Table II shows the complexities of a nonchecksum matrix multiplication performed in a systolic array and the redundancy required by the checksum technique.

Consider a band matrix multiplication with the checksum technique using the system shown in Fig. 6. Each column of processors in the array performs computations for the elements in a diagonal of matrix C. When a processor is faulty and located in the sth column from the left of the processor array, the elements $c_{\max(q_c-s+i,\,i),\,\max(i,\,s-q_c+i)}$,[4] $i = 1, \cdots, n$ are unreliable. For example, any faulty processor in the fourth column (the middle one) of the array in Fig. 6 will cause the diagonal with elements $c_{i,i}$ of the matrix C to be unreliable. The pattern of the erroneous matrix elements of a matrix is

[4] q_c is the number of diagonals which contain at least one nonzero element in the upper triangular matrix, including the main diagonal, of band matrix C.

TABLE II
THE COMPLEXITY OF A MATRIX MULTIPLICATION PERFORMED IN A SYSTOLIC ARRAY AND THE REDUNDANCY REQUIRED BY THE MULTIPLICATION SYSTEM WITH THE CHECKSUM TECHNIQUE

	The complexity of the matrix multiplication without fault-tolerance techniques	The redundancy required with the checksum technique	Redundancy[a] ratio
processor	$W1 * W2$	$W1 + W2$	$O(1/W1)$
time	$n + \min(W1, W2)$	$W1 + W2$	
adder	0	$3(W1 + W2) - 4$	
buffer	0	$\leq \log_2(W1) + \log_2(W2) + 2\min(W1, W2) + 2$	
TSC comparator	0	2	

[a]Assume $O(W1) = O(W2)$.

shown in Fig. 7(a); "?" shows the unreliable matrix elements. The value of s can be computed from the inconsistent row and column with the smallest indexes x and y

$$s = y - x + q_c.$$

Therefore, since the set of erroneous elements is located and each row of the matrix C consists of at most one erroneous element, the errors can be corrected.

In the matrix C_f, a fault in module $G1$ or $G3$ causes all rows or columns to be inconsistent. This means either that the erroneous elements are located on the row summation vector of the matrix C_f or that the module $G3$ is faulty. In these cases, by either regenerating the row summation vector or ignoring the inconsistency, the correct matrix C_f can be obtained. Similarly, the errors caused by a fault in module $G2$ or $G4$ can also be corrected. Fig. 7(b) and (c) shows the pattern of unreliable elements of the matrix C_f caused by a faulty module $G1$ or $G2$.

If we assume that the complexity of the adder, buffer, and TSC comparator is 10 percent of the complexity of the processor module (which contains a multiplier and other hardware), the redundancy ratios required by the checksum technique can be computed. For square matrices A and B having the same dimension n, Fig. 8(a) shows the redundancy ratio required for a mesh-connected processor array. Fig. 8(b) shows the redundancy ratio required for a matrix multiplication systolic array. These figures show that the redundancy ratio, in fact, asymptotically goes to zero for large matrices!

3) Lower Bound on the Number of Processors Required to Produce an Error-Detectable Checksum Matrix: It is obvious that the errors in a computation may not be detected when a single processor is used to produce a checksum matrix. In this section, we investigate the minimum number of processors required to produce a full checksum matrix in a matrix multiplication of a column checksum matrix with a row checksum matrix in which the errors caused by a faulty processor can always be detected.

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 will cause an inconsistent row or column which can be detected. However, when the errors can be connected to form a graph, as shown in Fig. 9, the errors may mask each other and cannot be detected. Such a graph is called a *loop* here and is defined as follows.

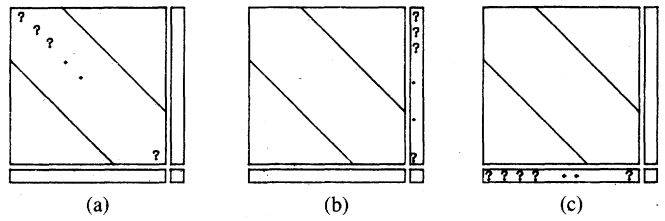

Fig. 7. The error patterns caused by (a) a faulty processor module, (b) a failure in the module G1, (c) a failure in the module G2.

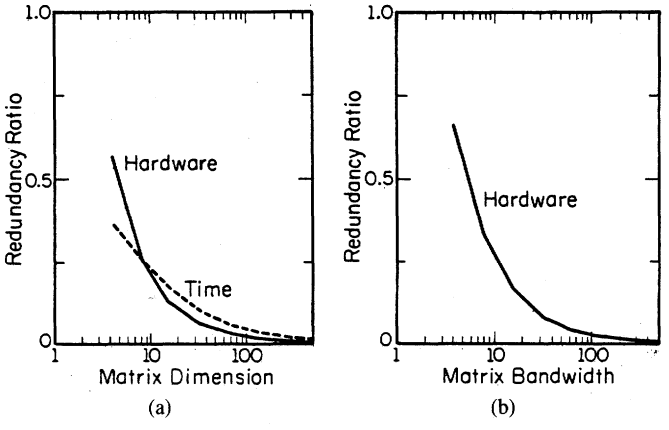

Fig. 8. The redundancy ratio required by the checksum technique for matrix multiplications using (a) a mesh-connected processor array, (b) a systolic array (assuming the adder, buffer, or comparator has 10 percent of the complexity of the processor, and the ratio of the execution time for a multiplication to that for an addition is 2).

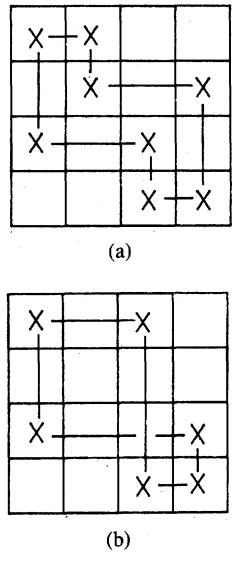

Fig. 9. The error-undetectable patterns in a full checksum matrix.

Definition 5.1: In an n-by-m ($n,m \geq 2$) chessboard with several *marked squares* (which are the squares with the "×" as shown in the example of Fig. 9), an *edge* is a connection between two marked squares in the same row or in the same column, a *path* is a sequence of edges, and a *loop* is a closed path (or cycle).

Lemma 5.1 and Theorem 5.1 show that when a processor performs the computations for $2n$ or more elements, these elements always form a loop.

Lemma 5.1: In an n-by-m chessboard ($n,m \geq 2$), when each row and column of the chessboard contains at least two marked squares, the set or a subset of those marked squares forms a loop.

Proof: Fig. 9(a) shows an example of a loop formed by marked squares when each row and column consists of at least two marked squares. Assume there does not exist a loop formed by the set or subset of the marked squares. This implies that there exists at least one marked square located on a row or a column which does not have any other marked square; this contradicts the assumption that all rows and columns contain at least two marked squares. Therefore, there exists at least one loop formed by those marked squares in the chessboard. □

Theorem 5.1: In an n-by-n chessboard ($n \geq 2$), for any given $2n$ marked squares, there exists a loop formed by these $2n$ squares or a subset of these $2n$ squares.

Proof: For an n-by-n chessboard with $2n$ marked squares, the rows and columns which consist of one or no marked squares cannot be used to form a loop; therefore, delete these rows and columns and let the number of remaining rows be n_r and the number of remaining columns be n_c, where $0 \leq n_r, n_c \leq n$.

The total number of squares in the n_r rows and n_c columns should be greater than or equal to the minimum number of remaining marked squares. Thus,

$$n_r * n_c \geq 2 * n - (n - n_r) - (n - n_c)$$

$$n_r * n_c \geq n_r + n_c. \tag{3}$$

Both n_r and n_c must be larger than 1 in order for (3) to be true. Therefore, according to Lemma 5.1, a loop exists formed by these $2n$ marked squares or a subset of these $2n$ squares. □

When the results produced by the same processor form a loop, the errors could mask each other. In order to guarantee that the errors caused by a single faulty processor can be detected in a matrix multiplication, according to Theorem 5.1, a processor cannot operate on more than $(2 * n - 1)$ elements in an n-by-n checksum matrix; therefore, the following theorem is true.

Theorem 5.2: At least $\lceil n^2/(2 * n - 1) \rceil$ ($n \geq 2$) processors are required to produce an n-by-n checksum matrix in which errors caused by a faulty processor can be detected. □

Let the number $\lceil n^2/(2 * n - 1) \rceil$ be denoted by $P_{\min}$. Fig. 10 shows examples of using $P_{\min}$ processors to produce elements in an n-by-n matrix where the elements performed by the same processor do not form a loop. The elements marked with the same number are assumed to be operated on by the same processor.

The number $n^2/(2 * n - 1)$ can be manipulated as shown in (4).

$$\frac{n^2}{2 * n - 1} = \frac{((n - 1/2) + 1/2)^2}{2(n - 1/2)} = n/2 + 1/4 + 1/(8 * n - 4). \tag{4}$$

1	2	1	3	1
2	1	2	1	3
1	2	3	2	2
2	1	3	3	1
1	3	3	2	3

(a) n=5, P_{min} =3

1	2	1	3	1	3
2	1	3	1	2	1
1	2	3	2	4	3
2	1	2	3	4	2
1	2	4	4	3	3
2	1	3	4	4	4

(b) n=6, P_{min} =4

Fig. 10. Arrangements using $P_{\min}$ processors to produce an error-detectable checksum matrix.

Equation (4) shows that $P_{\min}$ increases by 1 when n increases by 2. Equation (4) also implies that

$$P_{\min} = (n/2) + 1 \quad \text{if } n \text{ is even,} \tag{5}$$

$$P_{\min} = (n + 1)/2 \quad \text{if } n \text{ is odd.} \tag{6}$$

In the following, a procedure is given to extend the arrangement of using $P_{\min}$ processors for producing error-detectable n-by-n checksum matrices to using $P_{\min+1}$ processors for producing $(n + 2)$-by-$(n + 2)$ error-detectable matrices.

1) The arrangement for producing elements $c_{i,j}$, $1 \leq i, j \leq n$, is the same as the arrangement used to produce an error-detectable n-by-n checksum matrix.

2) Use the $(P_{\min+1})$st processor to perform the computations for the elements $c_{2i-1,n+1}$, $c_{2i,n+2}$, $c_{n+1,2i-1}$, $c_{n+2,2i}$, $c_{n+1,n+2}$, $c_{n+2,n+1}$, and $c_{n+2,n+2}$, where $i = 1, 2, \cdots, \lceil n/2 \rceil$.

3) Use the ith processor to perform the computations for elements $c_{2i-1,n+2}$, $c_{2i,n+1}$, $c_{n+2,2i-1}$, and $c_{n+1,2i}$, where $i = 1, 2, \cdots, \lceil n/2 \rceil$.

4) The other elements of matrix C_f are produced by the $P_{\min}$th processor.

From (5) and (6) and the above procedure, we know the $P_{\min}$th processor only performs the computation for $c_{n+1,n+1}$ when n is even and for $c_{n+1,n+1}$, $c_{n,n+2}$, and $c_{n+2,n}$ when n is odd. The arrangement of this extension is shown in Fig. 11, which shows that the elements computed by the $(P_{\min+1})$th processor do not form a loop, and any other processor only performs no more than one element in the $(n + 1)$th and $(n + 2)$th rows and columns; therefore, the errors can be detected. Fig. 10 shows arrangements of using $P_{\min}$ processors to produce error-detectable checksum matrices for the cases of an odd value of n and an even value of n. Thus, we see that the procedure can be recursively used to produce permutations for any value of n to use $P_{\min}$ processors to produce n-by-n error-detectable matrices in matrix multi-

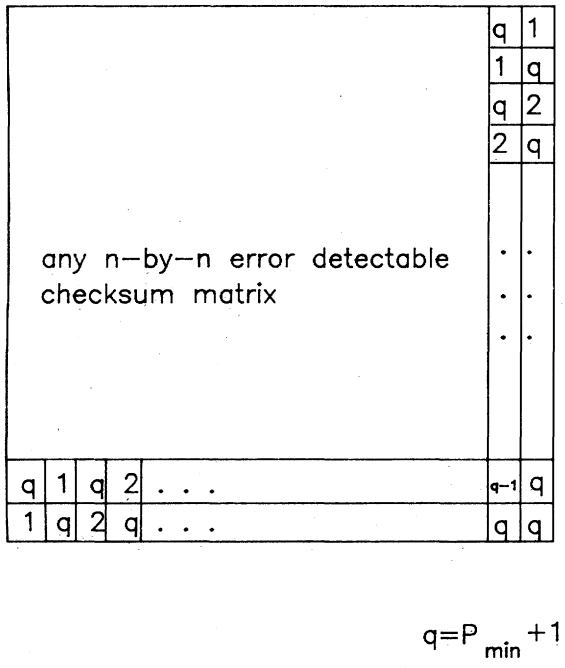

Fig. 11. P_{min+1} processors producing an $(n + 2)$-by-$(n + 2)$ error-detectable checksum matrix.

plication or addition. Also, we know that P_{min} is a tight lower bound of the number of processors required to produce an error-detectable checksum matrix.

B. Matrix LU Decomposition

From Section IV we know that a full checksum matrix C_f can be decomposed into a column checksum lower triangular matrix L_c and a row checksum upper triangular matrix U_r. The decomposition of an n-by-n full checksum matrix performed in an n-by-n mesh-connected processor array is described as follows.

1) Each element $c_{i,j}$ of the matrix C_f is loaded into a processor $P(i,j)$; let $c^1_{i,j}$, the value of $c_{i,j}$ in the first iteration, be $c_{i,j}$.

2) For $j = 1$ to n by 1

a) in the jth row $u_{j,k} = c^j_{j,k}$

b) in the jth column $l_{i,j} = c^j_{i,j}/u_{j,j}$ $\quad i,k > j$

c) $c^j_{i,k} = c^{j-1}_{i,k} - l_{i,j} * u_{j,k}$.

Next j.

(Row/column broadcast capability can be used here to help the data transfer.)

3) After Step 2), the processor $P(i,j)$ obtains the result $l_{i,j}$ when $i > j$ and $u_{i,j}$ when $i \leq j$.

4) The processor in the $(i + 1)$th row can be used to verify the checksum property in the ith row; similarly, the checksum property in each column can be verified without being masked by a faulty processor.

When a faulty processor $P(x,y)$ produces an incorrect result during the computation, the propagation of the error is described as follows.

a) The erroneous result is confined to an element $c_{x,y}$ before the $\min(x,y)$th iteration.

b) The value of $l_{x,y}$ and $u_{x,y}$ is obtained at the $\min(x,y)$th iteration; thus, the xth row and yth column will be the first inconsistent row and column, and the faulty processor is located at their intersection.

c) After the $\min(x,y)$th iteration, the error is propagated to all of the x' rows $(x' > x)$ and y' columns $(y' > y)$.

When a checksum band matrix is decomposed into two checksum matrices using a systolic array, the operation is similar to the one in [2]; however, redundant hardware is required to produce the summation vectors and to verify the checksum property.

The error propagation and the method for faulty processor location are similar to that of a decomposition performed in a mesh processor array; the faulty processor can also be located at the intersection of the first inconsistent row and the first inconsistent column in U_r and L_c, respectively.

C. Matrix Inversion

In this section, the checksum technique is applied for detecting errors concurrently when matrix inversion is performed in a processor array. The matrix inversion of a band matrix is usually a dense matrix except when the input matrix is a triangular matrix. Therefore, only the inversion of dense matrices is considered.

The matrix inversion of an n-by-n matrix can be performed using an n-by-$(2n + 1)$ mesh-connected array, as shown in Fig. 12. The left half n-by-n array stores the matrix A, the right half n-by-n array stores an identity matrix, and the last column stores the summation of the elements in the corresponding row. Such a row consisting of $2n$ information elements and their summation is called a checksum encoded vector (CSEV), as defined in Section IV. Then, the Gaussian elimination method can be used to eliminate the elements in the left half of the array (except for the diagonal elements) and to divide each row by the value of the nonzero element. An identity matrix is then obtained in the left half of the array and the inverse of matrix A is obtained in the right half of the array. The elimination operation is done by repeating (7) for $j = 1$ to $j = n$.

$$\text{Row}_i^{j+1} = \text{Row}_i^j - a^j_{i,j}/a^j_{j,j} * \text{Row}_j^j \quad \text{for } 1 \leq i \leq n \text{ and } i \neq j. \tag{7}$$

(Row_i^j means Row i in the jth iteration. Again, the row/column broadcast technique can be used to transfer the data efficiently during the operation.)

From Corollaries 4.1 and 4.2 and (7), we know that under the fault-free condition, the final information obtained in each row is a CSEV. When a faulty processor is located in the left half of the array, the errors will propagate through the array; however, the errors are not masked as long as the matrix A is nonsingular (a singular matrix does not have an inverse). When the faulty processor is located in the right half of the array or in the $(n + 1)$th column, the errors are confined to that column. Thus, the errors can be detected. A modification of the checksum technique can be used to correct errors during matrix inversion [19]; this is not discussed here due to lack of space.

D. The Error Coverage of the Checksum Technique Used with Uniprocessors

When the checksum matrix operations are performed in uniprocessor systems, a faulty processor could potentially cause all of the elements of the output matrix to be incorrect. In this section, we estimate the probability of detecting errors, and show that it is high.

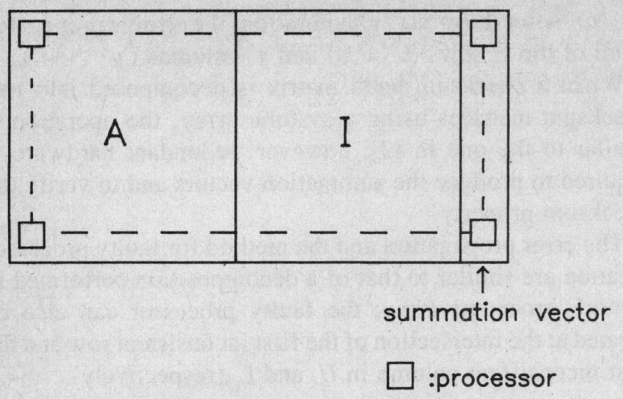

Fig. 12. Checksum matrix inversion performed in a processor array.

TABLE III
THE MINIMUM ERROR COVERAGE (DETECTABLE ERROR PERCENTAGE) OF CHECKSUM MATRIX OPERATIONS PERFORMED IN UNIPROCESSORS, WHEN THE ERRORS ARE GENERATED RANDOMLY

n	$P_{d\min}$ (percent)
1	75
3	93.75
5	98.4375
7	99.60937
9	99.90234
11	99.97558
13	99.99389
15	99.99847
20	99.99995

For a processor, we assume that a given defect causes d consecutive output bits to be unreliable and these bits are generated randomly. In a row or column of an $(n + 1) * (n + 1)$ full checksum matrix, the probability that the error in one element masks the sum of the errors in the other n elements is less than 2^{-d} since the errors are generated randomly. Thus, the probability of the undetectable errors in a column or a row p_u is no more than 2^{-d} and the probability of undetectable errors of a full checksum matrix P_u is no more than $(p_u)^{(n+1)}$. The worst-case value of p_u is ½ when $d = 1$; therefore, P_u is not larger than $2^{-(n+1)}$ and the probability of error detection of a full checksum matrix P_d is not less than $1 - P_u$. Table III shows the relation between $P_{d\min}$, the minimum detection probability, and n. It shows that the checksum technique also provides a high error detection capability when the computations are performed in uniprocessors.

VI. CONCLUSION

In this paper, the new technique of algorithm-based fault tolerance has been described and it has been applied to matrix operations. This technique is especially useful for fault tolerance in multiprocessor systems. The technique requires very low redundancy and covers a broad set of failures.

We hope that this concept of utilizing the information about the algorithm being performed will lead to new results in cost-effective fault-tolerance techniques for various applications.

ACKNOWLEDGMENT

We wish to thank the anonymous referees for their suggestions, especially Referee E for pointing out the work in [21].

REFERENCES

[1] K. E. Batcher, "Design of a massively parallel processor," *IEEE Trans. Comput.*, vol. C-29, pp. 836–840, Sept. 1980.
[2] H. T. Kung and C. E. Leiserson, "Algorithms for VLSI processor arrays," in *Introduction to VLSI Systems*, C. A. Mead and L. A. Conway, Eds. Reading, MA: Addison-Wesley, 1980, pp. 271–292, sect. 8.3.
[3] J. Von Neumann, "Probabilistic logics and the synthesis of reliable organisms from unreliable components," *Automata Studies*, no. 34, pp. 43–99, Princeton Univ. Press.
[4] J. G. Tryon, "Quadded logic," in *Redundancy Techniques for Computing Systems*, Wilcox and Mann, Eds. Washington, DC: Spartan, 1962, pp. 205–228.
[5] D. A. Anderson, "Design of self-checking digital networks using coding techniques," Coord. Sci. Lab., Univ. Illinois, Tech. Rep. R-527.
[6] D. A. Reynolds and G. Metze, "Fault detection capabilities of alternating logic," *IEEE Trans. Comput.*, vol. C-27, pp. 1093–1098, Dec. 1978.
[7] J. H. Patel and L. Y. Fung, "Concurrent error detection in ALUs by recomputing with shifted operands," *IEEE Trans. Comput.*, vol. C-31, pp. 589–595, July 1982.
[8] D. J. Lu, "Watchdog processors and structural integrity checking," *IEEE Trans. Comput.*, vol. C-31, pp. 681–685, July 1982.
[9] Y. Crouzet and C. Landrault, "Design of self-checking MOS LSI circuits, application to a four-bit microprocessor," *IEEE Trans. Comput.*, vol. C-29, pp. 532–537, June 1980.
[10] P. Banerjee and J. A. Abraham, "Fault characterization of VLSI MOS circuits," in *Proc. IEEE Int. Conf. Circuits Comput.*, New York, Sept. 1982, pp. 564–568.
[11] R. W. Hamming, "Error detecting and error correcting codes," *Bell Syst. Tech. J.*, vol. 29, pp. 147–160, Jan. 1950.
[12] J. J. Shedletsky, "Error correction by alternate-data retry," *IEEE Trans. Comput.*, vol. C-27, pp. 106–114, Feb. 1978.
[13] W. Peterson and E. J. Weldon, *Error-Correcting Codes*. Cambridge, MA: MIT Press, 1972.
[14] A. M. Patel and S. J. Hong, "Optimal rectangular code for high density magnetic tapes," *IBM J. Res. Develop.*, pp. 579–588, Nov. 1974.
[15] R. L. Burden, J. D. Faires, and A. C. Reynolds, *Numerical Analysis*. Prindle, Weber & Schmidt, 1979, p. 321.
[16] D. E. Knuth, *The Art of Computer Programming, Vol. 2*. Reading, MA: Addison-Wesley, 1969.
[17] P. Elias, "Error-free coding," *Trans. IRE*, vol. PGIT-4, pp. 29–37, 1954.
[18] J. H. Wilkinson, *Rounding Errors in Algebraic Processes*. Englewood Cliffs, NJ: Prentice-Hall, 1963.
[19] K. H. Huang, "Fault-tolerant algorithms for multiple processor systems," Ph.D. dissertation, Dep. Comput. Sci., Univ. Illinois, 1983.
[20] K. H. Huang and J. A. Abraham, "Efficient parallel algorithms for processor arrays," in *Proc. 1982 Int. Conf. Parallel Processing*, Bellaire, MI, Aug. 24–27, 1982, pp. 271–279.
[21] R. A. Willoughby, "Sparse matrix algorithms and their relation to problem classes and computer architecture," in *Large Sparse Sets of Linear Equations*, J. K. Reid, Ed. New York: Academic, 1971, pp. 257–277.
[22] J. A. C. Bingham, "A method of avoiding loss of accuracy in nodal analysis," *Proc. IEEE*, pp. 409–410, 1967.

Kuang-Hua Huang (S'80–M'83) received the B.S. degree in electrical engineering from National Cheng-Kung University, Taiwan, China, in 1973, the M.S. degree from the University of Kentucky, Lexington, in 1978, and the Ph.D. degree in 1983 from the University of Illinois, Urbana-Champaign, both in computer science.

He joined AT&T Technologies, Inc., Engineering Research Center, Princeton, NJ, in January 1983. His current research interests are testing, fault-tolerant computing, and high-performance computer architectures.

Jacob A. Abraham, for a photograph and biography, see this issue, p. 485.