Algorithm-Based Fault Tolerance on a Hypercube Multiprocessor

PRITHVIRAJ BANERJEE, SENIOR MEMBER, IEEE, JOE T. RAHMEH, CRAIG STUNKEL, V. S. NAIR, KAUSHIK ROY, VIJAY BALASUBRAMANIAN, AND JACOB A. ABRAHAM, FELLOW, IEEE

Abstract—**Hypercube multiprocessors have recently offered a cost effective and feasible approach to supercomputing through parallelism at the processor level by directly connecting a large number of low-cost processors with local memories which communicate by message-passing instead of shared variables. This paper discusses the design of a fault-tolerant hypercube multiprocessor architecture. Most of the recently proposed schemes of fault tolerance in parallel architectures address mainly the issue of reconfiguration of a parallel architecture once a faulty processor is identified. The schemes assume the existence of an off-line diagnosis strategy which locates the faulty processor. We propose the detection and location of faulty processors concurrently with the actual execution of parallel applications on the hypercube using a novel scheme of algorithm-based error detection. We have implemented system-level error detection mechanisms for three parallel applications on a 16-processor Intel iPSC hypercube multiprocessor: 1) matrix multiplication, 2) Gaussian elimination, and 3) fast Fourier transform. Schemes for other applications are under development. We have performed extensive studies of error coverage of our system-level error detection schemes in the presence of finite precision arithmetic which affects our system-level encodings. Finally, the paper proposes two reconfiguration schemes that allow us to isolate and replace faulty processors with spare processors. These schemes of reconfiguration are integrated with the error detection schemes to form a truly fault-tolerant hypercube multiprocessor.**

Index Terms—**Error coverage, hypercube multiprocessors, parallel algorithms, reconfiguration, system-level error detection.**

I. INTRODUCTION

A. Hypercube Multiprocessors

THE advent of cost-effective VLSI components in the past few years has made feasible the commercial development of massively parallel computers with hundreds of processors. Many different parallel architectures are under development, but one of the most commercially successful large-scale parallel architectures to date has been the Boolean hypercube [1]. Implementations of the hypercube architecture range from experimental prototype systems [2], to commercially available systems from Intel, Ametek, NCUBE, and Floating Point Systems.

A significant difference between hypercubes and most other parallel processors is that these machines use message-passing instead of shared variables for communication between concurrent processes. Each processor has a private local memory. This type of architecture is more readily scaled up to very large numbers of processors than multiprocessor designs based on globally shared memory. This model of parallel processing has some desirable characteristics with respect to error confinement as well. A faulty processor can be prevented from corrupting data in other processors if the faults are detected quickly. Contrast this to a shared memory multiprocessor where a faulty processor can potentially write into any location in memory and thereby corrupt an entire system within a very short time.

A hypercube multiprocessor consists of 2^N processors that are connected by direct links according to the binary N-cube interconnection pattern. Let us assume that the processors are consecutively numbered from 0 to $2^N - 1$. Each processor is directly connected to N other processors whose binary tags differ from its own by exactly one bit. Topologically, this arrangement places the processors at the vertices of an N-dimensional cube. Fig. 1 shows a four-dimensional hypercube, where the nodes represent processors and the edges represent direct communication links.

Since the probability of any one or more processors failing in such highly concurrent systems is quite large, it is desirable to build some fault tolerance feature into them. Fault-tolerant network architectures are therefore emerging as an important area of study [3]–[6]. In this paper, we will address the issue of fault tolerance in hypercube architectures.

B. Alternate Definitions of Fault Tolerance

We need to clarify the definition of fault tolerance in a multiple processor architecture before proposing our fault tolerance scheme. One definition used in the literature states that a network is fault tolerant (operational) as long as all the active processors are connected by fault-free links and processors [4]. This definition is appropriate for use in loosely coupled multiprocessors where the interconnection topology is not that important since tasks can be executed on multiple processors as long as they are interconnected. According to this definition, the hypercube architecture is $(N - 1)$-fault tolerant for a 2^N processor hypercube.

Manuscript received November 5, 1987; revised February 27, 1990. This work was supported in part by the National Science Foundation under Grants NSF MIP 86-57563 PYI, NSF MIP 86-19121, in part under Equipment Grant CCR 8705240, in part by the SDIO Innovative Science and Technology Office and managed by the Office of Naval Research under Contract N00014-88-K-0624, and in part by a Shell Doctoral Fellowship.

P. Banerjee, V. S. Nair, K. Roy, and V. Balasubramanian are with the Department of Electrical Engineering and the Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, Urbana, IL 61801.

J. T. Rahmeh and J. A. Abraham are with the Department of Electrical and Computer Engineering, University of Texas at Austin, Austin, TX 78712.

C. Stunkel is with the IBM T. J. Watson Research Center, Yorktown Heights, NY.

IEEE Log Number 9037191.

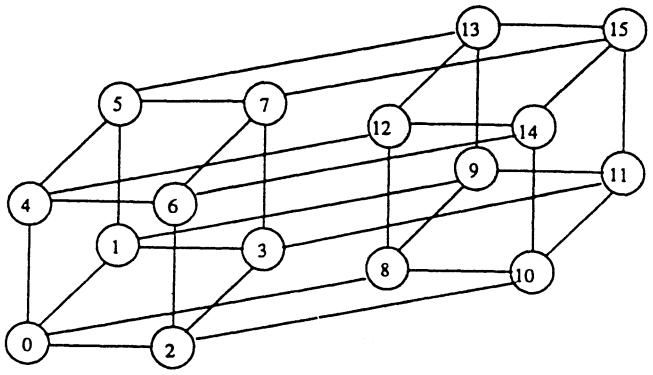


Fig. 1. Four-dimensional hypercube multiprocessor.

A second definition of fault tolerance used in the literature is more appropriate for tightly coupled microprocessors where the parallel applications running on the system depend on the interconnection topology. According to this definition, a network is fault tolerant if, under failure of one or more processors, a number of active processors is connected together according to the original topology of the network for which it was designed. According to this definition, the hypercube architecture is fault tolerant because, in the presence of a number of faulty processors, it is always possible to find a subset of fault-free processors that are still connected by the hypercube connection of a lower dimension. However, this view of fault tolerance in a hypercube results in a tremendous underutilization of resources. For example, if a single processor in a seven-dimensional (128-node) hypercube becomes faulty, the resultant working system would be a six-dimensional (64-node) hypercube, which reduces the performance by 50% even though less than 1% of the system is faulty.

In this paper, we are interested in designing a fault-tolerance hypercube architecture that will enable the realization of a hypercube topology of the original dimension (seven in the example) even under failures. This involves augmenting the topology of the hypercube using a number of spare processors and links. These issues are discussed in Section VI of this paper.

C. Fault Detection in Parallel Architectures

One issue that is commonly not rigorously addressed in the design of fault-tolerant parallel architectures is the mechanism for detecting faulty processors. Some researchers advocate the use of off-line testing of each processor in the hypercube assuming there is a set of functional tests that can be run by one processor on another [7]–[9]. It is widely agreed that it is quite difficult to validate the completeness of the functional testing strategies. Also, off-line testing can only detect permanent faults. It is well known that transient and intermittent failures occur more frequently than permanent failures in computer systems [10]. In order to detect these failures, it is necessary to have some kind of concurrent fault detection mechanisms.

High coverage concurrent fault detection can be achieved by implementing self-checking computers [11]. The techniques for doing that in medium performance computers involve SEC/DED codes in memory, error detecting codes on internal buses, and a mixture of coding and/or duplication of the processor and I/O logic. Although these techniques are well understood, there remains serious problems in implementing processing nodes with high-coverage concurrent fault detection. First, the high cost of detection in the processor and I/O logic is often unacceptable. Second, designers use commercially available chips, and chip manufacturers are not willing to sacrifice performance for fault tolerance features.

It is now generally agreed that there is a need for a hierarchical fault tolerance scheme, where certain faults are detected and recovered at lower levels; others are detected and recovered at a higher level [11]. In this paper, we also assume the existence of such a hierarchical scheme. We will propose some techniques for detecting faults at the system level.

D. Algorithm-Based Fault Tolerance

Recently, a new scheme for obtaining reliable results from computations using some on-line system-level checks of data has been proposed. Algorithm-based fault tolerance deals with a system-level method of achieving fault tolerance at low cost by tailoring the fault-tolerance scheme to the algorithm to be performed. This technique has been applied to matrix computations which form the basis of many computation-intensive tasks [12], [13]. The concept of algorithm-based fault tolerance has also been applied to algorithms for solving such problems as the fast Fourier transform [14], [15], matrix equation solvers [16], sorting, and evaluation of arithmetic expressions and polynomials [17], *QR* factorization, recursive least squares [18], and singular value decomposition [19]. Theoretical issues relating to algorithm-based fault tolerance have been reported in [20] and [21]. While these proposed techniques were interesting, none of the results were practically applicable since there were not many commercially available systolic array processors for actual evaluation of the schemes. The recent introduction of hypercube multiprocessors by several commercial vendors has provided an opportunity for actually evaluating the applicability of our algorithm-based fault tolerance schemes on a real machine, instead of a simplistic theoretical evaluation. We, therefore, decided to use a 16 processor Intel iPSC-2 hypercube as our testbed for investigation.

Numerous algorithms for computationally intensive tasks have been developed by researchers that are suitable for execution on hypercube multiprocessors [22]. One characteristic of many of these algorithms is that they are extremely structured. It is possible to apply algorithm-based techniques for concurrent error detection on the hypercube with very low overhead [23].

An issue that has not been rigorously addressed in any of the algorithm-based fault tolerance schemes in the past is the effect of finite precision arithmetic on the system-level encodings. In this paper, we have evaluated the effectiveness of the algorithm-based error detection mechanisms in the presence of finite precision arithmetic. We discuss the design of algorithm-based techniques for three parallel applications on a hypercube multiprocessor: 1) matrix multiplication, 2) Gaussian elimination, and 3) fast Fourier transform. Other applications are currently under investigation.

All the schemes described in the paper are in reality plausibility checks rather than being rigorous checks with complete

error coverage. However, in the presence of finite precision arithmetic, any high-level encoding defined on the data will have incomplete coverage even though it might be otherwise rigorous. The only way to get perfect error coverage is to use conventional techniques such as triplication and voting. In high performance systems such as hypercube multiprocessors, this approach is unacceptable. Our schemes offer a low-cost alternative to system-level concurrent error detection. The only way to determine the effectiveness of these schemes is through implementation on an actual machine and evaluating the error coverage for a large number of artificially created errors. This paper reports on the results of such a study. We have implemented these schemes on a commercially available hypercube multiprocessor from Intel. We have evaluated the coverage of our system-level error detection mechanisms by injecting both permanent and transient errors in the data paths.

It is appropriate at this point to clarify the notion of faults and errors as formally defined in [24]. A *fault* is a physical defect in the system, whereas an *error* is the manifestation of a fault at the logical level. As will be discussed in the next section, our methodology does not really determine the fault coverage but the error coverage of the algorithm-based schemes.

II. Methodology for Studying Error Detection Schemes

In this section, we will describe briefly the system on which we implemented the concurrent error detection mechanisms, and the methodology used to evaluate the error coverage and time overhead of our schemes.

A. Hypercube System

The testbed for our study was an Intel iPSC-2/D4/MX hypercube which consists of 16 processing nodes, each of which contains an 80386 CPU and 80387 floating point coprocessor, 4 megabytes of memory, and some direct routing hardware for message passing. Each node also contains eight bidirectional communication links managed by a direct-routing hardware. Seven of these links physically connect the nodes together and provide dedicated point-to-point communication. The eighth link provides direct access to and from the cube manager for program loading, data input/output, and diagnostics.

When using the iPSC, one implements a parallel processing application as a set of sequential processes that operate in parallel. The distribution of the input data to the node processes and the collection of the results are performed by a host process running on the cube manager. Each node process initializes, then waits for input data from the host, after receiving which it starts the computation. Node processes communicate by sending messages through the hypercube links. There is no concept of a shared data among processes.

B. Error Injection Scenario

One of the objectives of our study was to evaluate the error coverage of our algorithm-based error detection mechanisms. For each of the schemes to be described in the paper, a working parallel algorithm was written in C for the Intel iPSC-2 hypercube. One of the processors in the hypercube was then designated to be faulty at random. We then arbitrarily chose one of the modules within the processor, such as a multiplier, or an adder to be faulty. This is a reasonable assumption since in most current generation processor chips, the multipliers are implemented separately from the ALU (containing the adder). In this study, we will report on the results on only two types of modules, adders and multipliers; however, the results can be generalized easily to other types of modules such as dividers, square root units, CORDIC units, etc.

It should also be noted that errors in the data manipulation path also cover errors in memories or communication paths. An error during a memory read operation can be modeled as an incorrect input to a data manipulating module and will be reflected as an error produced at the output of the module. Similarly, an error during a memory write operation can be modeled as a correct write of an erroneous data produced by the module. Similarly, errors in the communication paths will show up as errors in data processing modules. It is therefore sufficient to investigate error detection for data manipulating modules. Faults in memory or communication paths that produce gross errors such as complete loss of data or control can be detected through time-out mechanisms. The other concern is what happens in case of control flow errors [25], [26]. The study of the detectability of control flow errors using algorithm-based techniques is a topic of further research and will not be addressed in this paper.

For each type of module, we wanted to simulate both permanent and transient faults, using software techniques, in order to study the detectability of errors produced in those modules. A permanent fault is one where the physical defect is present throughout the time interval of interest (in our case, the algorithm execution time). It should be noted, however, that the permanent existence of a defect does not guarantee the activation of the fault for a given input pattern. A transient fault is one where the defect is present over a short period within the interval of interest. We chose this time period to be about 10% of the total run-time of the algorithm. We also wanted to perform a comparative study of gross faults versus finer faults i.e., faults which produce an error in the entire output word versus those which change only a single bit at a time.

In order to perform a true fault injection, one needs to know the physical implementation of a module, for example, whether an adder is implemented as a ripple carry or a carry look-ahead. In the absence of such implementation details, we decided to inject errors instead. This is in compliance with the formal definitions of a fault and an error in [24]; a *fault* is a physical defect in the system, whereas an *error* is the manifestation of a fault at the logical level. On the basis of these discussions, four types of module errors were considered.

1) Permanent word error: Error injection was performed at the word level (by assigning a random word at the output of the faulty module), for about 50% of the total algorithm run-time, on the assumption that about half the time, a physical defect actually causes an error (is active).

2) Permanent bit error: This was simulated by changing a random bit value at the output, for about 50% of the total time. While the previous case models catastrophic faults, this case models faults whose effects are less severe.

3) Transient word error: Here, the word level error in-

jection was limited to around 5% of the total time. This is in accordance with our earlier assumptions that a transient fault is present in an interval measuring about 10% of the total run-time and that the fault activation rate is about 50%.

4) Transient bit error: The random bit change at the output was carried out for about 5% of the total time.

For each of the above errors and an error-free operation, we carried out a large number of simulations for different sets of input data in the presence of finite precision arithmetic for the 32-bit floating point representation. For example, for a permanent bit adder error, we injected an error (changed the bit value) at the node program in C, for every instance of addition in a particular processor; this was carried out for each of the 32 bits on ten different data sets. We then repeated the above experiment for a fault in a different processor. We, thus, performed a total of $32 \times 10 \times 2 = 640$ simulations for a permanent bit adder error.

C. Effects of Finite Precision Arithmetic

We now address the issue of finite precision arithmetic and its relation to error coverage. Many researchers have derived theoretical bounds on the errors in computation due to finite-precision arithmetic. The magnitude of a roundoff error in computation is limited by the wordlength; the shorter the wordlength, the larger the error. The norm of the error in computation can be written as $K^*F(N)^*2^{-2t}$, where K is an application specific constant, $F(N)$ is a function of the problem size that depends on the application, and t is the size of the mantissa in the floating point representation. For example, $F(N) = \log(N)$ for the fast Fourier transform [27].

In all our experiments, we kept the problem size and the mantissa size fixed for the normal and hardware-error-injected runs of the algorithms; hence, the norm of the error due to roundoff could be expressed as a constant K', which is chosen to maximize error coverage without *false alarms*. A false alarm is defined to be the incorrect identification of roundoff errors as hardware errors. The characterization of false alarms was experimentally carried out by determining the minimum constant K' such that for 50 different input data sets, no false alarms were observed. Having obtained an estimate of the error due to roundoff, we flagged an error as being due to hardware fault if the relative error between the final result and the correct result (=difference between the two results expressed as a percentage of the correct result) was more than the estimated roundoff error. We noted that many errors due to hardware faults were not detected using our on-line schemes when the constant was too high; conversely, the number of false alarms was high when the constant was too low. The *error coverage* that will be reported for various schemes in this paper is defined to be the percentage of all injected hardware errors that are detected by our schemes. It is worthwhile mentioning here that, in practice, some errors affecting the lower order bits might not change the output values by any appreciable amount; ignoring such errors could lead to an increase in the resultant error coverage.

D. Time Overhead

Another objective of our study was to evaluate the realistic time overheads of our error detection mechanisms. In the past, researchers proposing schemes for algorithm-based fault tolerance have analyzed the time overheads of their schemes very approximately by counting the extra steps (such as multiply and add) in the modified algorithms. No efforts were made to estimate realistic times including the effects of computation, interprocessor communication, and input/output. In this paper, we have performed detailed studies of exact time overheads required in our schemes.

Some of the performance features of the hypercube are given below. Each processing node which contains an 80386 CPU is rated to operate at 4 MIPS; the 80387 coprocessor is rated at 300 KFLOPS; the message routing uses wormhole routing techniques and takes about 350 μs for short messages and 1 ms for 1 kilobyte messages. The message delay increases very slightly with the length of the message. These considerations influenced certain decisions during our algorithm designs.

We make some brief comments about the timing measurements on the hypercube. We measured the actual run-times for the algorithms with and without the system-level checks for each application and estimated the actual time overheads. The Intel iPSC-2 hypercube is a single user machine with no virtual memory; hence, the measurements were not affected by extraneous factors such as page faults, context swaps, and system load which create problems during measurements on most general purpose computer systems. The MCLOCK command of the Intel hypercube operating system has a granularity of several milliseconds, since the internal hardware clock timer is updated at that frequency. Hence, whenever we wanted to estimate the run-time for one application on the hypercube, we performed the measurement for ten repeated instances of the application, and divided the total time by 10.

In the following sections, we report on our proposed concurrent error detection mechanisms for three applications for the hypercube multiprocessor and present the results of our experimental evaluation of the error coverage and the time overheads.

III. Matrix Multiplication

A. Algorithm Description

We are interested in performing the multiplication:

$$C = A \times B$$

where C, A, and B are full matrices of sizes $p \times r$, $p \times q$, and $q \times r$, respectively. Numerous algorithms have been proposed in the literature for performing matrix multiplication in the hypercube, each having its own characteristic of speedup and memory usage. We used a scheme which requires more memory per node than other schemes but has much better properties with respect to error confinement and less interprocessor communication.

The host partitions the matrix A into a number of rectangular strips by rows equal to the number of processors in the hypercube and sends one strip to each processor. The complete matrix B is also sent to each processor. Each processor P_i performs the submatrix multiplication $C_i = A_i \times B$ using a sequential algorithm. At the end, each processor sends the submatrices of the result matrix back to the host.

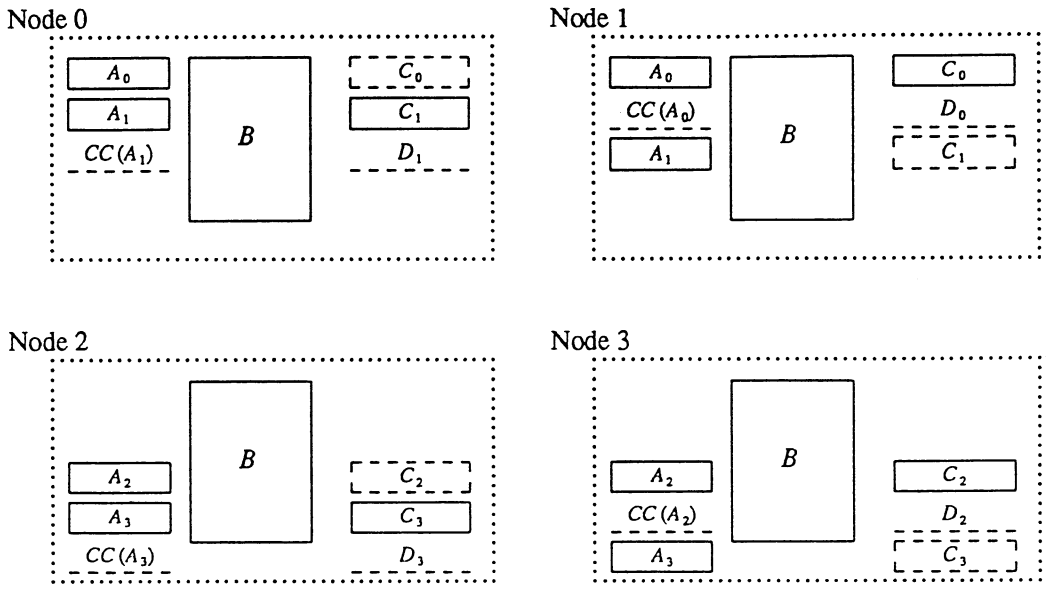


Fig. 2. Matrix multiplication on a four-processor hypercube.

The algorithm is modified to include system-level checks based on the checksum matrix encoding proposed in [12]. The checksum matrix encoding scheme consists of the following steps.

1) The elements of each column of the matrix $A_{p\times q}$ are summed together. The result is the row $CC(A)_{1\times q}$ called the column checksum of A.

2) The matrix multiplications $C = A \times B$ and $D = CC(A) \times B$ are performed.

3) The column checksum of matrix C, $CC(C)$, is computed and compared to D. In the absence of hardware errors and roundoff errors, $CC(C)$ and D should be identical.

We incorporate the above steps (at the strip level) into the matrix multiplication algorithm for the hypercube as follows.

1) The processors of the hypercube, hereafter referred to as nodes, are logically divided into $n/2$ pairs (a node with the even address i is paired with the node with address $i+1$). We will refer to the nodes in a pair as mates. The host sends the matrix B to all the nodes, and partitions the matrix A into rectangular strips (by row) sending strip A_i to node i and its mate.

2) Node i computes $C_i = A_i \times B$ and sends C_i to the host.

3) Node i computes the column checksum of the A strip of its mate, $CC(A_{\text{mate}(i)})$, and then the D strip of its mate, $D_{\text{mate}(i)} = CC(A_{\text{mate}(i)}) \times B$.

4) Node i obtains the C strip of its mate, $C_{\text{mate}(i)}$.

5) Node i checks its mate by comparing $CC(C_{\text{mate}(i)})$ to $D_{\text{mate}(i)}$ and sends the results (pass or fail) to the host.

6) The host judges the computation error-free if all nodes "pass," otherwise, it judges the computation erroneous.

The above procedure is illustrated for a four-processor hypercube in Fig. 2 where the items shown with dashed lines are computed locally at the node containing them and the items shown with continuous lines represent those that are received by the processor. For example, node 0 will receive strip A_0, strip A_1, and the matrix B. It will compute $C_0 = A_0 \times B$, $CC(A_1)$, and $D_1 = CC(A_1) \times B$. It will then receive C_1 from node 1, compute $CC(C_1)$ and compare it to D_1.

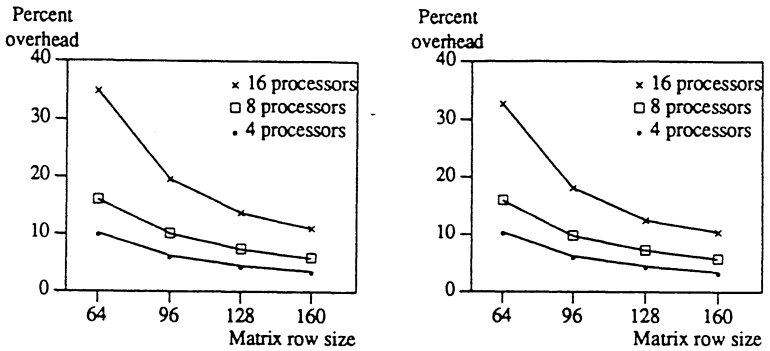
If there is a fault in a node during the regular matrix multiplication computation, it will produce an error that will be detected by the row check with a high probability (we will quantify this later in the paper), since the checksum row for a strip C_i is calculated by the node that is the neighbor of node i. Note that the above algorithm serves to detect the presence of a faulty node but not to locate it. For example, if node 0 were faulty, then node 1 will check its result and declare it faulty; at the same time, node 0 will check node 1 and decide, being faulty, that node 1 is also faulty. The host, getting two faulty indications, will judge the computation erroneous but will not be able to locate the faulty node. We extended the above algorithm to achieve fault location by performing two phases of checking. The first phase is identical to what has been described so far. The second phase is similar to the first except that each node is paired with a mate different from the one of phase one. For example, node 0 is paired with node 1 in phase one, and node 2 in phase 2. This way each node gets two diagnostics from two different nodes. The node that is diagnosed to be faulty twice is labeled as the faulty node.

B. Experimental Results

We ran our experiments on an example multiplication of two 64×64 matrices on hypercubes of 4, 8, and 16 processors. We measured the time required to perform a normal matrix multiplication and the time required to perform the multiplication using system-level checks. Table I summarizes the results of error coverage for 32-bit and 64-bit floating point arithmetic. We make the following observations from the table. First, the coverage of word-level errors compared to bit-level errors is much higher since the corresponding magnitude of the error is much higher. Second, the error coverage is much higher when 64-bit precision is used as opposed to 32-bit precision. This implies that the algorithm-based error detection techniques using system-level encodings should be used in the presence of high-precision arithmetic. This result should not simply be viewed as a limitation of our error detection schemes but an indication of how the results of com-

TABLE I
PERCENT ERROR COVERAGE IN FULL CHECKING OF MULTIPLICATION OF TWO 64×64 MATRICES

	32-bit precision			64-bit precision		
	Number of processors			Number of processors		
	4	8	16	4	8	16
Transient bit error/adder	78.91	78.83	75.78	91.41	89.06	84.77
Transient bit error/multiplier	72.66	69.53	67.19	89.06	85.55	80.86
Transient word error/adder	100.00	100.00	100.00	100.00	100.00	100.00
Transient word error/multiplier	100.00	100.00	100.00	100.00	100.00	100.00
Permanent bit error/adder	86.72	82.03	82.03	94.92	92.19	88.28
Permanent bit error/multiplier	78.13	75.00	75.00	91.02	88.28	83.98
Permanent word error/adder	100.00	100.00	100.00	100.00	100.00	100.00
Permanent word error/multiplier	100.00	100.00	100.00	100.00	100.00	100.00



double precision single precision

Fig. 3. Execution time overhead of the matrix multiplication error detection scheme.

putations can be affected by finite precision arithmetic. One interesting thing to note is the decrease in the error coverage for increasing number of processors for a given problem size. This is due to the fact that each processor does less computations (and therefore is less affected by errors) as the number of processors increases.

Fig. 3 shows the overhead in execution time incurred by the fault detection scheme for a range of (square) matrix and cube sizes. The figures show that for a fixed size hypercube, the time overhead can be quite high for low problem sizes (e.g., 35% for a 64×64 matrix multiplication on a 16 processor hypercube), but decreases with increasing problem sizes. This is due to the high relative communication delays for smaller granularity of computations. For a fixed size hypercube, if the size of the problem increases, each processor has to do more computations since the size of the strips is larger. The number of communication steps is fixed by the size of the hypercube. The cost of a communication step is not that different for a small matrix as opposed to a large matrix since the message delays in the Intel hypercube are relatively independent of the size of the message. For a fixed problem size, if the size of the hypercube is increased, the amount of normal computation that each processor has to do decreases, while the computation for the checks increases, hence the overhead increases. For large problem sizes, the overheads become less than 15%, which makes our schemes attractive.

IV. GAUSSIAN ELIMINATION

A. Algorithm Description

When a set of linear equations $A\tilde{x} = \tilde{b}$ is solved for a single right-hand side $\tilde{b}$, the solution is usually obtained by performing Gaussian elimination to create an upper triangular matrix. This step is followed by either obtaining a diagonal A matrix through Gaussian elimination of the upper triangle, or by back substitution, which solves for the value of x_n, and substitutes this value into the x_{n-1} equation to solve for x_{n-1}, etc. On a hypercube, this Gaussian elimination method is often solved by distributing complete rows of $A\tilde{x} = \tilde{b}$ to each hypercube processor node [28]. Pivoting in linear equations algorithms provides for stable (no division by zero) and more numerically precise solutions. Although full pivoting by row and column provides the most numerically precise solutions, some form of partial pivoting is most often used because of the reduced overhead for calculating the next pivot. The movement of rows due to partial pivoting can be expressed as a vector $\tilde{p}$, where p_i represents the physical row corresponding to logical row i. Using this convention, actual row swapping need not take place. Instead, elements of the matrix are indirectly addressed using this vector $\tilde{p}$. For the hypercube implementations in the succeeding sections, this use of indirect addressing will be assumed.

With the linear system distributed by rows, column checksums can be used to detect any errors caused by a single node. Given a 2^N node hypercube, each $2^N - 1$ rows in the system of n linear equations is augmented by a row of column checksums. Each group of the resulting 2^N rows will be referred to as a *submatrix* of the full system. These checksums make it possible (assuming infinite precision) to detect an arbitrary number of errors in a single node. However, row checksums are also needed to positively identify which node is failing once an error is flagged. In this paper, one row checksum per row is assumed, which adds an extra column to the system of

linear equations. Once any column checksum fails, the corresponding rows of the *submatrix* are analyzed to find the first row checksum failure. The corresponding node is the failing node.

A general method of handling pivoting in the presence of column checksums is now developed that requires an extension of the weighted checksum definition vector given in [13], which defines a coded vector as follows:

$$D = (a_1 a_2 \cdots a_n \, \mathrm{WCS}_1 \, \mathrm{WCS}_2 \cdots \mathrm{WCS}_r),$$

where a_i is a data element, and WCS_i is defined as

$$\mathrm{WCS}_i = (a_1 a_2 \cdots a_n) \cdot (W_{i1} W_{i2} \cdots W_{in}).$$

A more useful checksum can be defined that will be shown to provide more efficient and precise results on column weighted checksum vectors when pivoting is being performed. A weight is given to the checksum value itself, and the sum of this weighted checksum and the weighted data are defined to be zero. To simplify the discussion, a single weighted checksum WCS will be assumed, and WCS will be defined by this equation:

$$W_0 \, \mathrm{WCS} + (a_1 a_2 \cdots a_n) \cdot (W_1 W_2 \cdots W_n) = 0.$$

The checksum element is now no different than any data element, and, in particular, the checksum element can be allowed to participate in pivoting. Pivoting involving the checksum element will cause the checksum element to become a data element for the purposes of Gaussian elimination. This operation is legitimate only because the full checksum row is a linear combination of all of the data rows of the full matrix, and thus it is linearly dependent upon the data rows. The only problem to avoid is allowing an entire *submatrix* into the final n rows of the linear system at one time. This situation will result in one linearly dependent row, which will destroy the Gaussian elimination process.

Prudent choices of weights for the checksums can result in reduced computation time for pivoting and row elimination. The weights for the *unity* checksum WCS are defined as

$$W_i = 1 \text{ for all } i.$$

For this weighting, no weights need to be kept with the data. Pivoting simply updates a permutation vector $\tilde{p}$, instead of physically causing rows to swap. $\tilde{p}$ is then used as an indirect address mapping logical rows to physical rows. This technique is commonly used in Gaussian elimination.

The column checksum is not as straightforward to implement as the row checksum. This follows from the fact that row subtractions are being performed, and these row subtractions do not preserve the natural column checksum properties. Special adjustments must be made to assure that the checksum satisfies the checksum equation. Consider the initial matrix representing $A\tilde{x} = \tilde{b}$, and, for simplicity, assume that no pivoting is ever required. To further simplify the discussion, assume that $n < 2^N$, so that column checksums are weighted summations of the *entire* column.

$$\begin{array}{cccccc} a_{11} & a_{12} & \cdots & a_{1n} & b_1 \\ a_{21} & a_{22} & \cdots & a_{2n} & b_2 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} & b_n \\ \mathrm{WCS}_1 & \mathrm{WCS}_2 & \cdots & \mathrm{WCS}_n & \mathrm{WCS}_b. \end{array}$$

To preserve the properties of the weighted column checksums, it will be necessary to leave the elements of the pivot column that are below the pivot row unchanged. This is common practice anyway, since these elements serve no useful purpose in Gaussian elimination after being used in the pivot column. With this assumption, the first iteration of the elimination will produce the following matrix:

$$\begin{array}{cccccc} a_{11} & a_{12} & \cdots & a_{1n} & b_1 \\ a_{21} & a_{22} - \frac{a_{12}}{a_{11}} a_{21} & \cdots & a_{2n} - \frac{a_{1n}}{a_{11}} a_{21} & b_2 - \frac{b_1}{a_{11}} a_{21} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{n1} & a_{n2} - \frac{a_{12}}{a_{11}} a_{n1} & \cdots & a_{nn} - \frac{a_{1n}}{a_{11}} a_{n1} & b_n - \frac{b_1}{a_{11}} a_{n1} \\ \mathrm{WCS}_1^1 & \mathrm{WCS}_2^1 & \cdots & \mathrm{WCS}_n^1 & \mathrm{WCS}_b^1 \end{array}$$

where WCS_j^k represents the value of WCS_j after the kth iteration (ignoring pivoting). For the pivot column, there is no change to the data, so $\mathrm{WCS}_k^1 = \mathrm{WCS}_k$. For $j > 1$ (nonpivot columns), WCS_j can be computed as follows:

$$\mathrm{WCS}_j^1 = \mathrm{WCS}_j - \frac{a_{1j}}{a_{11}} \mathrm{WCS}_1 - W_1 a_{1j}.$$

For succeeding iterations of Gaussian elimination, this equation generalizes to (for iteration p and $j > p$):

$$\mathrm{WCS}_j^p = \mathrm{WCS}_j^{p-1} - \frac{a_{pj}^{p-1}}{a_{pp}^{p-1}} \mathrm{WCS}_p^{p-1} - \sum_{k=1}^{p} W_k a_{kj}^{p-1}.$$

It is not desirable to compute this summation, but it can be removed by adding variables. Two variables are used to keep track of each WCS_j. One variable (WCA_j) will hold the weighted summation of the first p elements in the column (where p = the current iteration), and the other variable (WCB_j) will hold the weighted summation of the rest of the data elements in the column. The WCA_j variables are initialized to zero, and the WCB_j variables are initialized to the weighted checksum as before. All WCA_j and WCB_j variables remain unchanged for column $j \leq p$ (disregarding pivoting). With these changes, the general update equations for $j > p$ now become

$$\mathrm{WCA}_j^p = \mathrm{WCA}_j^{p-1} - W_p a_{pj}^{p-1}$$

$$\mathrm{WCB}_j^p = \mathrm{WCB}_j^{p-1} - \frac{a_{pj}^{p-1}}{a_{pp}^{p-1}} \mathrm{WCB}_p^{p-1}.$$

This checksum update is more costly than the normal data element update. However, the simple *unity* checksum removes the multiplication from the WCA_j calculation.

Now pivoting will be integrated into this column checksum approach. WCA values represent weighted sums above the current pivot line, and therefore will never switch during a pivot operation, even if the WCB row is involved in the pivot. If a checksum row is being switched with a data row, all that is necessary is to switch the data row with WCB. This simply involves updating the $\tilde{p}$ permutation vector as always. These observations can be readily seen by examining the checksum equations. Assuming that data row p and checksum row 1 are involved in the pivoting, the equation before pivoting is

$$W_0(\mathrm{WCA}_j^k + \mathrm{WCB}_j^k) + \sum_{i=1}^{n} W_i a_{ij}^k = 0.$$

After pivoting, WCB_j^k and a_{pj}^k are switched, but the equation still holds.

This completes the necessary theory for handling the updates of the column checksums during the elimination of the lower triangle of the system $A\tilde{x} = \tilde{b}$.

The rest of the computation is back substitution. Column checksum equations are of no use during this phase of the computation. On the hypercube, this back substitution occurs somewhat sequentially, since one node k_1 will perform the full calculation of x_i given the previously calculated values $x_{i+1}, \cdots, x_n$. After this calculation is complete, this node will send x_i to the other nodes. Since this is a largely sequential computation, an idle neighboring node k_2 can check the x_i calculation by performing the identical calculation, provided that node k_2 also contains row i. Rather than passing this row i from node k_1 to node k_2, this implementation takes advantage of the fact that every row has been passed to every node during the forward elimination phase. A subset of the hypercube connections provides a simple ring network, and each node k_i in this ring saves every pivot row sent by its predecessor node k_{i-1}. The resulting x_i check is *exact*, since the two separate calculations must yield an identical answer.

Performing the actual check of a checksum on a hypercube is not a trivial task. For row distribution in a hypercube, since every node contains data elements from each column, all nodes remain involved in computation throughout every iteration. With the full row distribution method, the selection of the pivot row is a cooperative effort between all of the nodes. Each node contains elements of the pivot column, and these elements are successively compared to other elements via a tree-like communication pattern that terminates at node 0. This communication takes N steps. This global comparison results in the chosen pivot row being collected at node 0, and node 0 then broadcasts this result to all other nodes. At the same time that this pivot calculation takes place, the checksum calculation and the pivot checking can be done using the elements of the pivot column. The pivot calculation is checked through pure redundancy—each node receives enough information from the proceeding node to repeat the proceeding node's section of the pivot calculation. The pure redundancy is slight in cost compared to the normal communication overhead required for the pivot calculation. These pivot column elements are also weighted and successively added to form column checksums for each submatrix, and node 0 flags any errors found in the column checksums.

TABLE II
PERCENT ERROR COVERAGE IN GAUSSIAN ELIMINATION OF A 100×100 MATRIX

	32-bit precision			64-bit precision		
	Number of processors			Number of processors		
	4	8	16	4	8	16
Transient bit error/adder	81	78	75	92	88	86
Transient bit error/multiplier	78	78	75	91	86	88
Transient word error/adder	100	100	100	100	100	100
Transient word error/multiplier	100	100	100	100	100	100
Permanent bit error/adder	88	84	81	95	94	92
Permanent bit error/multiplier	84	81	78	94	92	91
Permanent word error/adder	100	100	100	100	100	100
Permanent word error/multiplier	100	100	100	100	100	100

Errors in a column checksum trigger the checking of the row checksums for its associated submatrix. Checking of the row checksums is accomplished with the same communication pattern as the back substitution checking. The hypercube configures itself as a ring, and each node passes information to its neighbor in the ring. This neighbor sums the data elements in a row and compares it to the actual row checksum.

B. Experimental Results

We ran our experiments on an example Gaussian elimination of a 100×100 matrix on hypercubes of 4, 8, and 16 processors. Table II summarizes the results of error coverage for 32-bit and 64-bit floating point arithmetic. It can be seen that the coverage for transient bit errors is quite low, but the coverage for the other error types is reasonably high. Also, it is obvious that 64-bit precision is desirable for our algorithm-based checking schemes.

We next measured the time required to perform a normal Gaussian elimination and the time required to perform the computation using system-level checks. Fig. 4 displays the execution overhead for various sizes of matrices and hypercubes. Note that this overhead decreases as both matrix size and cube dimension increase. The reasons are similar to those described for matrix multiplication.

V. FAST FOURIER TRANSFORM

A. Algorithm Design

The discrete Fourier transform is one of the most important computations in digital signal processing. It is used in computing the convolution of two signals, in spectral estimation, and numerous other applications. The discrete Fourier transform is given by

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

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 equation as

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

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

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

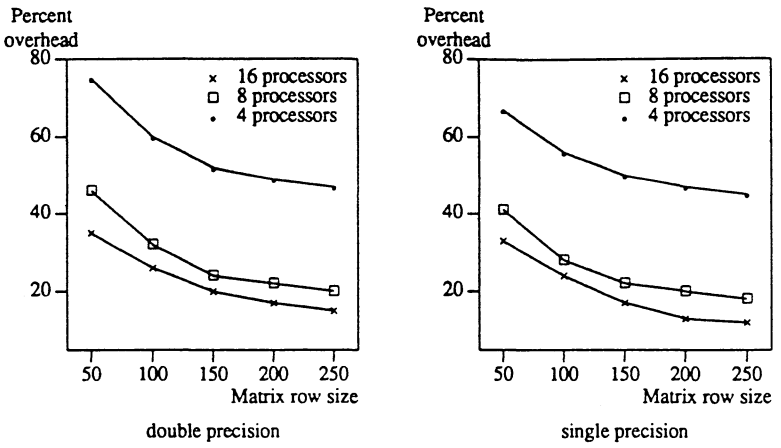


Fig. 4. Execution time overhead of Gaussian elimination error detection scheme.

The above equation is the basic equation for the FFT. In designing a parallel algorithm for the hypercube, the following assumptions were made.

1) The number of points is a power of 2, e.g., 2^P.

2) The number of processing nodes is a power of 2, e.g., 2^N. This assumption is satisfied by the hypercube topology.

3) The number of points is more than or equal to the number of nodes.

The parallel algorithm is described at a high level 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 a permuted manner.

y = bit_reversal_permutation (x)
for $i = 0$ **to** $2^N - 1$
  send $y[i^*2^{P-N} \cdots (i+1)^*2^{P-N}]$ to node i.
endfor.

Alternatively, all the points are broadcast to all the nodes and each node keeps the points it needs. This is slightly faster than what is described above.

c) The processing nodes will perform the iterative sequential FFT algorithm on the points they have received. Nodes will perform the computations independently and in parallel.

d) After they have computed the FFT of the points they were given, the nodes will exchange messages.

for $i = 0$ to $N - 1$
  neighbor=complement bit i of *id*
  if ($id < neighbor$)
    exchange the second half of node *id* with first half of *neighbor*
  else
    exchange the first half of node *id* with second half of *neighbor*
  perform the butterfly operation
endfor.

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

Fig. 5 illustrates how the algorithm works for a 16-point FFT on four nodes. Note that the vector y is the bit reversal permutation of the input vector x.

Several schemes of algorithm-based fault tolerance have been proposed for the FFT on butterfly networks [14], [15], [18]. We now state two properties of the fast Fourier transform which are used as checks in our hypercube algorithm.

Property 1 (Parseval's Theorem): The product of the number of points in the FFT and the sum-of-squares of the input points equals the sum-of-squares of the output points. This property was used in [18] for fault detection in an FFT butterfly network.

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 inputs to the sum of squares of the outputs of intermediate FFT computations. For example in Fig. 5, the results of the sum of squares of inputs to an 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.

Our error detection mechanism involves the following steps (SOS refers to sum of squares).

1) Each node computes the 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 output 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 comparing 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 that the fault was a permanent fault.

We assume that the results of intermediate computations are saved as checkpoints among the processing nodes in our parallel algorithm. In Fig. 5, for example, the results of the four-point FFT's after the first stage of the computations are available in each of the four nodes. These intermediate results

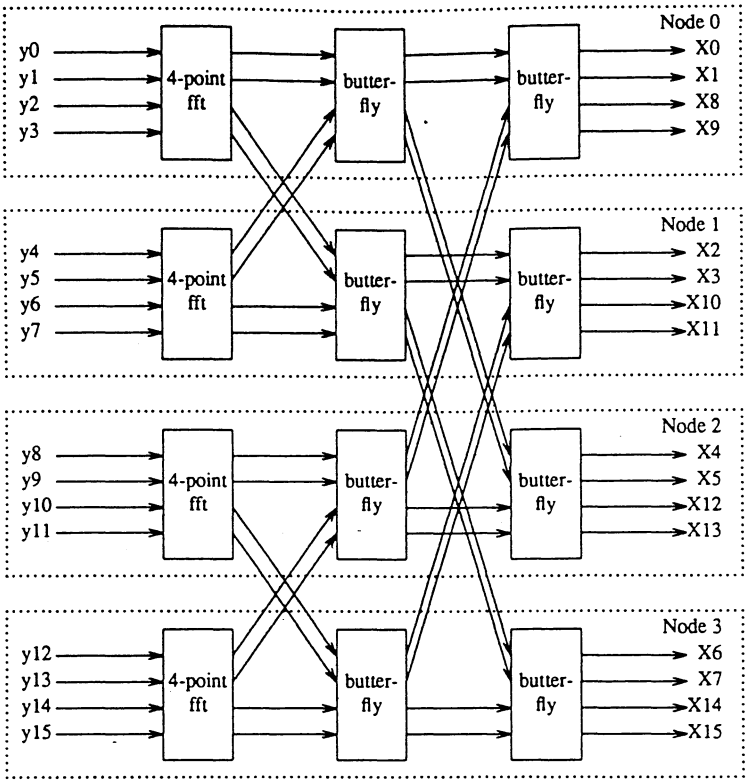


Fig. 5. 16-point FFT on a four-processor hypercube.

are checked using the same property. For example, the sum of squares of the inputs, $x(0)$, $x(4)$, $x(8)$, and $x(12)$, multiplied by 4, should equal the sum of squares of the four-point FFT's computed by node 0. These computations are checked by node 0 itself and its two neighbors, namely nodes 1 and 2. The results of the checks are sent to the host. Since 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 computations.

The next step is to verify the correctness of the second stage of computations, namely the eight-point FFT's. The inputs and outputs of a portion of the eight-point FFT, namely points $0, 1, 4, 5$, are sent by node 0 to two neighbors, nodes 1 and 2. Nodes 0, 1, and 2 independently compute the sum of squares and send three sets 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 (Property 2 of FFT). If an error is determined at this stage, the faulty processor is located to within two processors, e.g., nodes 0 and 1, or nodes 2 and 3.

The next step is to verify the correctness of the third stage of computations, namely the 16-point FFT. In this stage, the inputs and outputs of the points $0, 1, 8, 9$ are sent from node 0 to its two neighbors, 1 and 2. Again, the nodes perform the intermediate sum of squares of the 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). A discrepancy at this stage can locate the faulty processor to within four processors, which, in this example, is the entire hypercube.

At this stage, the host assumes that the fault is transient and repeats the entire set of computations. If the fault is permanent, then the same syndromes will be repeated. It is hoped that in case of a permanent fault, the fault will be detected in the earlier stages where the faulty processor is precisely identified, or is located to within a few processors. The reconfiguration strategy described in Section VI can be applied and the set of computations repeated assuming one of the suspect processors to be faulty. If the guess is incorrect, an alternate suspect processor is replaced using our reconfiguration strategy.

B. Experimental Results

We ran our experiments on an example 256-point FFT on hypercubes of 4, 8, and 16 processors. We measured the time required to perform a normal FFT and the time required to perform the FFT using system-level checks. Table III summarizes the results of error coverage for 32-bit and 64-bit floating point arithmetic. Fig. 6 shows the execution time overhead incurred by the error checking detection for 32-bit and 64-bit floating point arithmetic, a range of cube dimensions, and a range of problem sizes.

VI. Reconfiguration Techniques

In the preceeding sections, we have discussed some techniques for system-level concurrent fault detection. The first time a processor is identified as faulty, it is assumed to be a transient failure, and the computation is repeated. If the same failure occurs more than once on the same set of inputs, the fault is diagnosed to be a permanent fault. In that case, the faulty processor needs to be isolated from the system and a spare processor brought instead. We now address the issue of placing spare processors and links in an augmented hypercube topology such that in the presence of faulty processors,

TABLE III
PERCENT ERROR COVERAGE IN 256-POINT FFT

	32-bit precision			64-bit precision		
	Number of processors			Number of processors		
	4	8	16	4	8	16
Transient bit error/adder	85.16	82.03	75.00	94.53	91.41	88.67
Transient bit error/multiplier	77.34	72.66	71.88	90.63	85.55	85.94
Transient word error/adder	100.00	100.00	100.00	100.00	100.00	100.00
Transient word error/multiplier	100.00	100.00	100.00	100.00	100.00	100.00
Permanent bit error/adder	89.06	86.72	82.81	96.09	94.92	91.80
Permanent bit error/multiplier	85.16	83.59	79.69	96.09	94.14	92.19
Permanent word error/adder	100.00	100.00	100.00	100.00	100.00	100.00
Permanent word error/multiplier	100.00	100.00	100.00	100.00	100.00	100.00

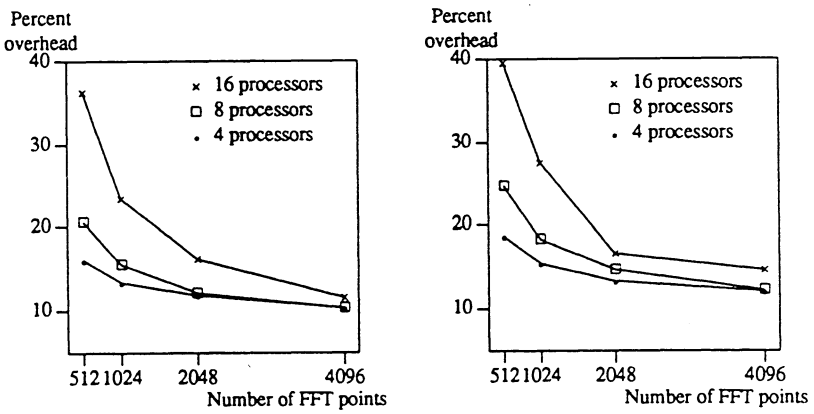


Fig. 6. Execution time overhead of the FFT error detection scheme.

a hypercube topology containing fault-free processors can be configured. In the augmented topology consisting of spare processors and links it is important that the degree of the nodes does not increase exponentially. The number of redundant processors should also be minimal.

Recently, Rennels has proposed a scheme for spare reconfiguration in hypercube topologies [6]. The processors in an N-dimensional hypercube having N normal hypercube ports are augmented by an additional port in the $(N+1)$st dimension. The reconfiguration scheme uses VLSI crossbar switches to bring in a spare processor using the additional dimension when a processor fails anywhere within the hypercube. For a hypercube containing 128 processors, each spare processor connects to eight VLSI crossbar chips each of which connect to 16 ports of a 16-processor subcube. Effectively, the degree (number of ports) of the spare processor node becomes equal to the size of the hypercube = 128. The scheme has also been extended to provide spares for each subcube of processors to reduce the degree of the spare node.

In this section, we propose two schemes for reconfiguration, one for reconfiguring from faulty links and processors, the other for reconfiguring from faulty links alone. Both the schemes minimize the number of communication ports of each processor.

A. Reconfiguration for Processor Sparing

The first scheme is designed to reconfigure faulty processors. We use two spare processors for every eight normal processors of the original hypercube topology. Each normal

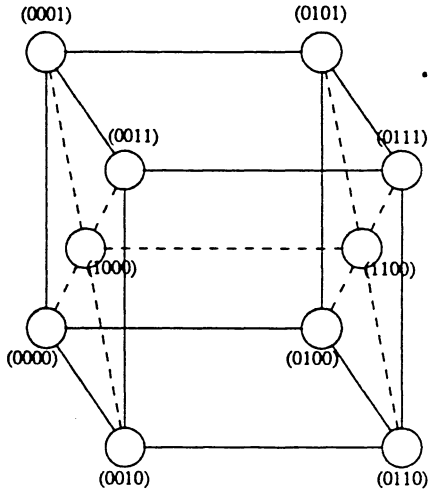


Fig. 7. Augmented three-dimensional hypercube topology (two spares).

processor node has a degree of $N+1$ and the degree of each spare node is $N+2$. The details of the connections of such a cube are shown in Fig. 7. Since we add redundant nodes, we increase the dimension of the cube by one. The address of a normal node for an eight-node fault-tolerant cube is determined by appending a 0 to the most significant bit position of the original 3-bit address whereas the address of a spare node is determined by appending a 1 to the original 3-bit address of any one of the normal processors connected to it. The only constraint is that there can be only one bit difference between the addresses of two spare nodes belonging to the same basic subcube. The addresses of nodes (normal/spare) can be de-

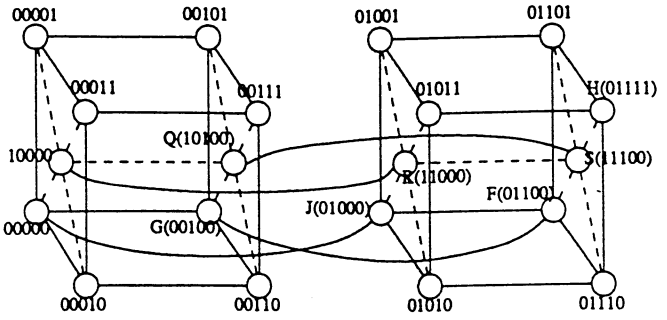


Fig. 8. Augmented four-dimensional hypercube topology (four spares).

termined for any dimension higher than three as usual. Fig. 8 shows a four-cube connection with four spare processors. It can be observed that the spare processors have been connected to themselves in a two-cube fashion. Therefore, the regularity of the structure is maintained, even when we increase the dimension.

When the failure of a normal processor is detected, the processor is replaced by a spare processor. The algorithm for replacement is given below.

1) Replace the faulty processor with the spare processor connected to it.

2) The link from the spare processor to the faulty processor and the link diagonally opposite to it are disabled. All other spare links are enabled.

3) The spare processor S which replaces the faulty processor sends its address and the faulty processor's address to all the spare processors connected to it.

4) Each spare processor connected to S performs a bitwise Exclusive-OR of its own address with the address of S and the address of the faulty processor F to determine the addresses of the nodes to which S should be connected.

As an example, let us consider the fault-tolerance four-cube shown in Fig. 8, where node $F(01100)$ is faulty. The reconfiguration steps are given below. Replace processor $F(01100)$ with the spare processor $S(11100)$. The links FS and HS are disabled. All other links are enabled. Spare processors R and Q are adjacent to processor S. Processor S sends addresses of $F(01100)$ and $S(11100)$ to both $R(11000)$ and $Q(10100)$. R calculates the address of the node it should be connected to ($J(01000)$ in this case) by EXORing (bit by bit) the addresses of S, R, and F. Similarly, Q EXOR's the addresses of S, Q, and F to determine its neighbor ($G(00100)$ in this case). Here, spare nodes R and Q act as switches.

B. Reconfiguration for Link Sparing

The second scheme for reconfiguration is designed to tolerate link failures alone. We assume that link failures are detected by coding techniques on the messages transmitted. The augmented topology is as follows.

Let us assume that the 2^N processors in the hypercube are addressed by their binary representations, $a_{N-1}a_{N-2}\cdots a_0$. The hypercube topology implies that the processor $a_{N-1}\cdots a_i\cdots a_0$ is connected to processor $a_{N-1}\cdots a_i\cdots a_0$ in the ith dimension. We propose that the processor be also connected directly to processor $\overline{a_{N-1}}\,\overline{a_{N-2}}\cdots\overline{a_0}$, i.e., complementing all the bits in the binary representation. This implies that the degree of each node is increased by one. The resultant topology is shown in Fig. 9.

Let us suppose that a link in dimension i of the hypercube becomes faulty. Then, the reconfiguration algorithm is as follows. All the dimension i links are disabled and the auxiliary links are activated instead. The processors are newly addressed as follows. The processor with old address $a_{N-1}\cdots a_{i+1}0a_{i-1}\cdots a_0$ has the same new address. However, the processor with old address $a_{N-1}\cdots a_{i+1}1a_{i-1}\cdots a_0$ is assigned the new address $\overline{a_{N-1}}\cdots\overline{a_{i+1}}1\overline{a_{i-1}}$. In other words, the processors whose binary addresses have a zero in the ith position, maintain the same addresses; however, the processors that have a one in the ith position will complement all the bits except the ith bit in their new addresses. It can be easily shown that under the new addressing scheme, the hypercube connections are still maintained but the processors switch roles. Fig. 9 shows the old and new addresses of the processors after the faulty links marked by crosses are replaced by spare links.

VII. Conclusions

Hypercube multiprocessors have recently offered a cost-effective and feasible approach to supercomputing through parallelism at the processor level by connecting a large number of low-cost processors with direct links. However, as systems become more complex, the probability of the failure of the systems increases unless specific measures are taken to tolerate faults within the system. This paper has discussed the design of a fault-tolerant hypercube multiprocessor architecture. Most of the earlier schemes of fault tolerance in parallel architectures addressed mainly the issue of reconfiguration of a parallel architecture once a faulty processor is identified. The schemes assumed the existence of an off-line diagnosis strategy which locates the faulty processor.

In this paper, we have proposed the detection and location of faulty processors concurrently with the actual execution of parallel applications on the hypercube using a novel scheme of algorithm-based error detection. We have implemented system-level error detection mechanisms for three parallel applications on a 16-processor Intel iPSC hypercube multiprocessor: 1) matrix multiplication, 2) Gaussian elimination, and 3) fast Fourier transform. Schemes for other applications are under development.

All the schemes described in the paper are in reality plausibility checks rather than being rigorous checks with complete

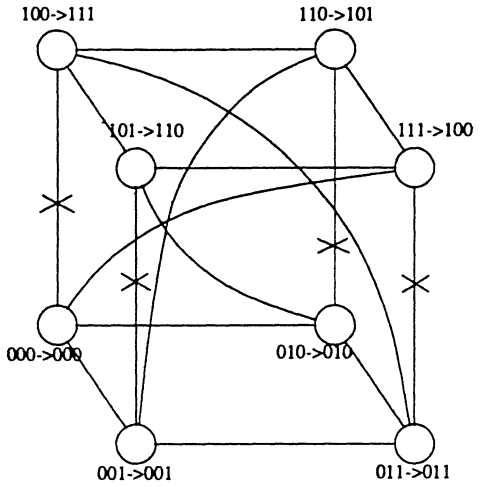


Fig. 9. Reconfiguration strategy for sparing links.

error coverage. However, in the presence of finite precision arithmetic, any high-level encoding defined on the data will have incomplete coverage even though it might be otherwise rigorous. The only way to get perfect error coverage is to use conventional techniques such as triplication and voting which are extremely costly. Our schemes offer a low-cost alternative to system-level concurrent error detection. The only way to determine the effectiveness of these schemes is through implementation in an actual machine and evaluating the error coverage for a large number of artificially created errors. We have implemented these schemes on a commercially available hypercube multiprocessor from Intel. We have evaluated the coverage of our system-level error detection mechanism by injecting both permanent and transient errors in the data paths. We have shown that we can achieve between 85–100% error coverage using our schemes with no modification to the existing hypercube architecture.

Finally, the paper proposes two reconfiguration schemes that allow us to isolate and replace faulty processors with spare processors. These schemes require some modification to the original architecture. These schemes of reconfiguration are integrated with the fault detection schemes to form a truly fault-tolerant hypercube multiprocessor.

References

[1] C. L. Seitz, "The Cosmic Cube," *Commun. ACM*, pp. 22-33, Jan. 1985.

[2] J. C. Peterson, J. Tuazon, D. Lieberman, and M. Pniel, "The Mark III hypercube-ensemble concurrent computer," in *Proc. 1985 Parallel Processing Conf.*, Aug. 1985, pp. 71-73.

[3] I. Koren, "A reconfigurable and fault-tolerant VLSI multiprocessor array," in *Proc. 8th Int. Symp. Comput. Architecture*, Minneapolis, MN, May 1981, pp. 425-442.

[4] D. K. Pradhan, "Fault-tolerant multiprocessor link and bus network architectures," *IEEE Trans. Comput.*, pp. 33-45, Jan. 1985.

[5] R. Negrini, M. Sami, and Stefanelli, "Fault tolerance techniques for array structures used in supercomputing," *IEEE Comput. Mag.*, pp. 78-87, Feb. 1986.

[6] D. A. Rennels, "On implementing fault tolerance in binary hypercubes," in *Proc. 16th Int. Symp. Fault-Tolerant Comput.*, Vienna, Austria, July 1986, pp. 344-349.

[7] J. G. Kuhl and S. M. Reddy, "Fault diagnosis in fully distributed systems," in *Proc. 11th Int. Symp. Fault-Tolerant Comput.*, June 1981, pp. 100-105.

[8] J. R. Armstrong and F. G. Gray, "Fault diagnosis in a Boolean *n*-cube array of microprocessors," *IEEE Trans. Comput.*, vol. C-30, pp. 587-590, Aug. 1981.

[9] E. Dilger and E. Ammann, "System level self-diagnosis in *n*-cube connected multiprocessor networks," in *Proc. 14th Int. Symp. Fault Tolerant Comput.*, Kissimmee, FL, June 1984, pp. 184-189.

[10] R. K. Iyer and D. J. Rossetti, "Permanent CPU errors and system activity: Measurement and modeling," in *Proc. Real-Time Syst. Symp.*, 1983.

[11] D. A. Rennels, "Fault tolerant computing—Concepts and examples," *IEEE Trans. Comput.*, vol. C-33, pp. 1116-1129, Dec. 1984.

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

[13] J. Y. Jou and J. A. Abraham, "Fault-tolerant matrix operations on multiple processor systems using weighted checksums," *SPIE Proc.*, Aug. 1984.

[14] ——, "Fault tolerant FFT networks," in *Proc. 15th Int. Symp. Fault Tolerant Comput.*, Ann Arbor, MI, June 1985, pp. 338-343.

[15] M. Malek and Y. H. Choi, "A fault-tolerant FFT processor," in *Proc. 15th Fault-Tolerant Comput. Symp.*, Ann Arbor, MI, June 1985, pp. 266-271.

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

[17] P. Banerjee and J. A. Abraham, "Fault-secure algorithms for multiple processor systems," in *Proc. 11th Int. Symp. Comput. Architecture*, Ann Arbor, MI, June 1984, pp. 279-287.

[18] A. L. N. Reddy and P. Banerjee, "Algorithm-based fault detection for signal processing applications," *IEEE Trans. Comput.*, 1990.

[19] C.-Y. Chen and J. A. Abraham, "Fault-tolerant systems for the computation of eigenvalues and singular values," *Proc. SPIE, Advanced Algorithms Architectures Signal Processing*, vol. 696, pp. 228-236, Aug. 1986.

[20] P. Banerjee and J. A. Abraham, "Bounds on algorithm-based fault tolerance in multiple processor systems," *IEEE Trans. Comput.*, vol. C-35, pp. 296-306, Apr. 1986.

[21] ——, "Concurrent fault diagnosis in multiple processor systems," in *Proc. 16th Fault Tolerant Comput. Symp.*, Vienna, Austria, July 1986, pp. 298-303.

[22] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, and J. K. Salmon, in *Solving Problems on Concurrent Processors*. Englewood Cliffs, NJ: Prentice-Hall, 1989.

[23] C. Aykanat and F. Ozguner, "A concurrent error detecting conjugate gradient algorithm on a hypercube multiprocessor," in *Proc. 17th Int. Symp. Fault-Tolerant Comput.*, Pittsburgh, PA, July 1987, pp. 204-209.

[24] J.-C. Laprie, "Dependable computing and fault tolerance: Concepts and terminology," in *Proc. 15th Annu. Symp. Fault-Tolerant Comput.*, June 1985, pp. 2-11.

[25] M. Schuette and J. P. Shen, "Processor control flow monitoring using signatured instruction streams," *IEEE Trans. Comput.*, vol. C-36, pp. 264-276, Mar. 1987.

[26] A. Mahmood and E. J. McCluskey, "Concurrent error detection using watchdog processors—A survey," *IEEE Trans. Comput.*, pp. 160-174, Feb. 1988.

[27] C. J. Weinstein, "Roundoff noise in floating point fast Fourier transform computation," *IEEE Trans. Audio Electroacoust.*, vol. AU-17, pp. 209-215, Sept. 1969.

[28] G. A. Geist and M. T. Heath, "Matrix factorization on a hypercube multiprocessor," in *Proc. SIAM 1st Conf. Hypercube Multiprocessors*, Knoxville, TN, Aug. 1985.

Prithviraj Banerjee (S'82-M'84-SM'90), for a photograph and biography, see the April 1990 issue of this TRANSACTIONS, p. 446.

Joe T. Rahmeh was born in Becharre, Lebanon, in 1960. He received the B.S., M.S., and Ph.D. degrees in electrical and computer engineering from the University of Illinois, Urbana, in May 1981, January 1984, and October 1988, respectively.

Currently he is an Assistant Professor at the University of Texas at Austin. His research interests include computer architecture and electronic computer-aided design.

Craig Stunkel received the B.S. and M.S. degrees in electrical engineering from Oklahoma State University in 1982 and 1983, respectively, and the Ph.D. degree in electrical engineering from the University of Illinois, Urbana, in 1990.

He is currently a Research Staff Member at IBM's Thomas J. Watson Research Center, Yorktown Heights, NY. From 1983 to 1986 he was employed with IBM, Rochester, MN, where he participated in the design of the AS/400. In 1986, he took an educational leave of absence to attend the University of Illinois. While at Illinois he held a Shell Doctoral Fellowship and a Research Assistantship. His current research interests include parallel architectures, algorithms, and performance analysis.

Dr. Stunkel is a member of the Association of Computing Machinery and the IEEE Computer Society.

V. S. Nair, for a photograph and biography, see the April 1990 issue of this TRANSACTIONS, p. 435.

Kaushik Roy received the B.Tech degree in electronics and electrical communications engineering from the Indian Institute of Technology, Kharagpur, in 1983.

He is currently a Ph.D. degree candidate at the University of Illinois at Urbana-Champaign. Since 1986, he has been a Research Assistant at the Coordinated Science Laboratory at the University of Illinois. In addition, he has worked at the Semiconductor Processing and Design Center of Texas Instruments, Dallas. His research interests include VLSI/computer-aided-design, fault-tolerant computing, and computer architecture.

Mr. Roy is a member of Phi Kappa Phi and a student member of the Association for Computing Machinery.

Vijay Balasubramanian, for a photograph and biography, see the April 1990 issue of this TRANSACTIONS, p. 446.

Jacob A. Abraham, (S'71-M'74-SM'84-F'85), for a photograph and biography, see the April 1990 issue of this TRANSACTIONS, p. 435.