Evaluation of a Fault Tolerant Distributed Broadcast Algorithm in Hypercube Multicomputers

Jace W. Krull
IBM Corporation
Boca Raton, FL 33431

Jie Wu
Department of Computer Science and Engineering
Florida Atlantic University
Boca Raton, FL 33431

Andres M. Molina
IBM Corporation
Boca Raton, FL 33431

Abstract

This paper performs a detailed evaluation of a fault-tolerant distributed broadcasting algorithm for cube connected networks. The main areas of evaluation are the following: (1) Algorithm effectiveness in the presence of multiple faults, (2) Establishing the maximum number of link faults allowed, before the algorithm fails to guarantee 100% effectiveness. The evaluation was done to networks connected in 3-, 4-, 5-, and 6-cube configurations. The results of the simulation were analyzed to establish algorithm characteristics under multiple faults.

Introduction

As the popularity and use of on-line computer systems has increased in recent years, so has the demand for greater system performance, and greater reliability of these systems. The demand for greater performance has led the computer community to the area of distributed and parallel systems architectures.

In the mid 70's, Tandem was exploring the uses of parallel systems as a way of providing fault tolerance. Since that time, a number of parallel systems architectures have been proposed (hypercube, tree, torus, and mesh) [1] that fulfill the demand for greater performance and fault tolerance.

Hypercube is one of many parallel systems architectures that have been proposed over the years [5]. As in all other parallel systems, it supports three basic types of interprocessor communication: one-to-one (unicast), one-to-many (multicast), and one-to-all (broadcast) [2]. Hypercubes are loosely coupled parallel processors based on the binary n-cube network and introduced under different names (cosmic cube, n-cube, binary n-cube, etc.). An n-cube parallel processor consists of 2^n identical processors, each provided with its own memory, and interconnected with n neighbors. A node address can be represented as:

$$I = I(n-1)\ldots I(i+1),\ I(i),\ I(i-1)\ldots I(1),\ I(0)$$

with $I(i) = 0$ or 1

The address of its neighboring nodes (n in all) can be represented as

$$I = I(n-1)\ldots I(i+1),\ \overline{I(i)},\ I(i-1)\ldots I(1),\ I(0)$$

Communication in hypercubes is achieved by message passing, whereby data and/or code are transferred from processor A to processor B by travelling across a

sequence of nearest neighbor nodes starting at processor A and ending in processor B.

Architecturally hypercubes exhibit many advantages, such as: simplicity of design, low cost, excellent mapping capabilities, and a unique ability to exploit particular topologies of problems or algorithms in order to minimize communication costs. Hypercubes also provide the ability to generate fault tolerant systems by simply shutting down failing nodes or by system reconfiguration [6].

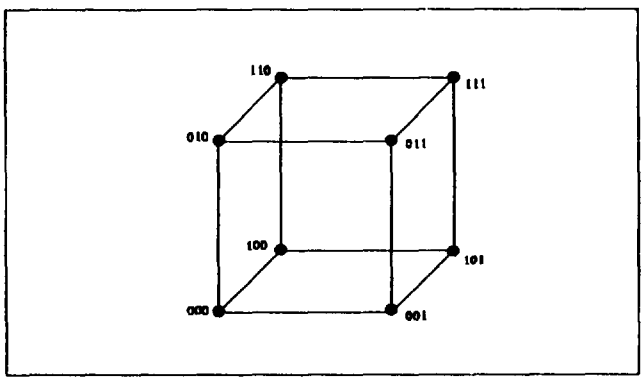

Figure 1. 3-D view of a 3 dimensional hypercube.

Fault-tolerant routing is one important issue in the design of a hypercube system. This paper will discuss the area of broadcasting in hypercubes, specifically fault-tolerant broadcasting. With few exceptions [7] [8], the area of fault-tolerant broadcasting in distributed systems has been neglected, mainly due to its complexity and cost (record keeping). Wu and Fernandez proposed an algorithm that uses the topological properties of hypercubes to guarantee rerouting with minimum additional traffic and without the need for a table of global information [3].

In this paper we intend to evaluate the fault coverage of Wu and Fernandez' algorithm, especially in cases of multiple faults. The evaluation considers hypercubes with dimensions range from 4 to 6 (8 - 64 nodes).

This paper is organized as follows: Section 2 of the paper provides a brief introduction of the algorithm. Section 3 presents a description of the software model written to simulate the algorithm. The results of the simulation are presented and analyzed in Section 4. Finally, Section 5 contains a summary of the results of the evaluation.

Algorithm Description

The algorithm is based on the concept of a binomial tree [3], [4]. A binomial tree is a tree which is in class B(k) for some k; the integer k is called the index of such a binomial tree. A class B(k) is recursively defined as follows:

1. Any tree consisting of a single node, a B(0) tree.
2. Suppose that Y and Z are disjoint B(k-1) trees for K > = 1. Then the tree obtained by adding an edge to make the root of Y become the left-most offspring of the root of Z is a B(k) tree.

The algorithm states that it can regenerate a faulty subtree, induced by a faulty node, through one of the leaves in the binomial tree with minimum regeneration traffic, and without node duplication. The type of faults considered were limited to fail-stop; a link either works properly or stops completely.

For simplicity, let us assume that distributed fault detection is achieved (for possible implementation of distributed fault detection see [3]). Once a fault is detected, the fault handler (node detecting the fault) disconnects the faulty subtree making it impossible to broadcast messages to the nodes in the faulty subtree. A faulty subtree is defined as the fault handler's descendants through the faulty link. The fault handler will then send a regeneration message that has the following format:

```
{ Broadcast Message, PCS, t }
```

The Path Coordinate Sequence (PCS) is defined Figure 2

Where direction is defined as the bit position that is different between two adjacent nodes in a binomial tree. For example, in Figure 4, the direction between node 000 and node 100 is 2.

The I(PCS) is called the regeneration path defined by PCS with a starting address I. I(PCS) is recursively defined as shown in Figure 3.

```
Path Coordinate Sequence

  =  {s,t}      for s > t & d ¬= s

  =  {s,t,s}    for s < t or d = s

Where:

  s = set of link directions leaving
      the fault handler excluding t
  t = direction entering the faulty
      subtree (link at fault)
  d = direction entering the fault handler
```

Figure 2. Path Coordinate Sequence definition

```
I(PCS)

  =  I                                  If L(PCS)  =0

             F(PCS)      R(PCS)
  =  I --> I       --> I                If L(PCS) ¬=0
                                        &  R(PCS) ¬=0

             F(PCS)
  =  I --> I                            If L(PCS) ¬=0
                                        &  R(PCS)  =0

Where:  F(PCS) = first(PCS)
        L(PCS) = length(PCS)
        R(PCS) = rest(PCS)
```

Figure 3. Regeneration Path Definition

Three operations: First, Rest, Length on PCS = {C(1), C(2), ... C(n)} are defined as follows:

```
First (PCS)  = C(1)

Rest (PCS)   = {C(2), C(3), ... C(n)}

Length (PCS) = 1 + Length(Rest(PCS)),
               with Length ({})=0
```

For instance if we have a binomial tree with a fault between node 001 and 011 as shown in Figure 4, the fault handler is node 001 and the corresponding values for s, t, and d are as follows:

$$s = \{ 0,2 \} \quad t = 1 \quad d = 0$$

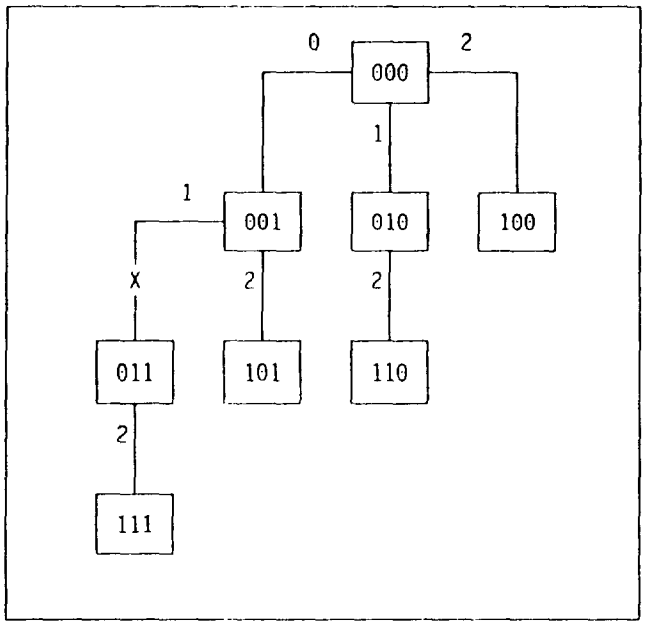

Figure 4. Binary tree with fault between node 001 and 011

and the two choices for Path Coordinate Sequence (PCS) are

$$PCS = \begin{cases} \{ 2,1 \} \\ \{ 0,1,0 \} \end{cases}$$

In this example the regeneration path (starting from the fault handler) is $001 \rightarrow 001^2 = 101 \rightarrow 101^1 = 111$ before the regeneration of the faulty subtree.

The fault handler will then send the regeneration message along the first link direction given in PCS. In our example let the PCS be { 2,1 }, then the fault handler will send the regeneration message through link direction 2. Each node receiving a regeneration message will pass it along through the direction given by the first member of the PCS, and it will also delete that member from the PCS. If that link does not exist as it is in the case of our example (node 5 link 1 or node 5 link to node 7 does not exist), then the new link is generated. If a node receives a regeneration message with the PCS equaling the empty set, then that node and subsequent nodes will regenerate the faulty subtree based on the following rules:

```
FOR i > t and i < n
   Generate a node in direction i
END
```

Where n equals the dimension of the hypercube (3 for our example)

Figure 5 is the resultant tree based on the regeneration algorithm.

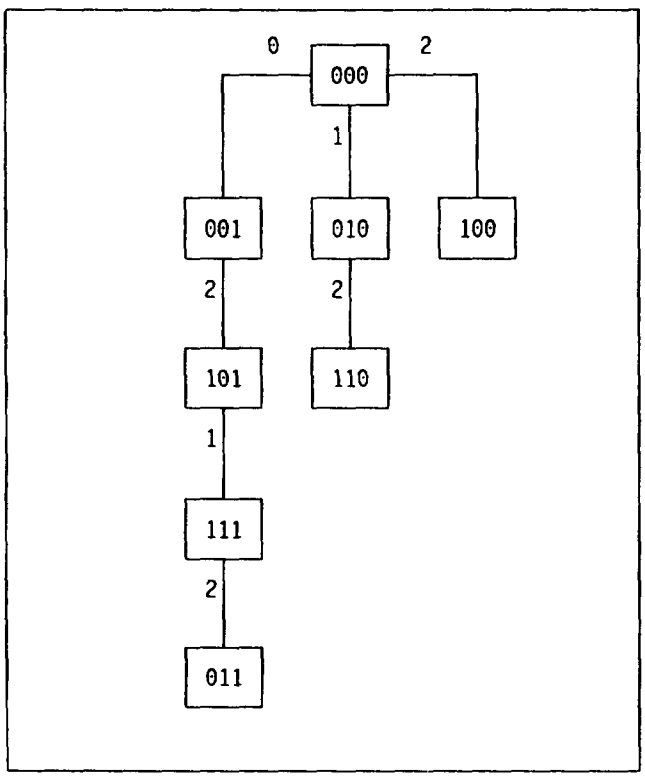

Figure 5. Regenerated tree

Normally the PCS = {s,t} has a higher priority than the PCS = {s,t,s} because of shorter regeneration paths. The PCS = {s,t,s} will be selected only when there are faults on the regeneration path(s) defined by PCS = {s,t}.

It has been proved [3] that following the above path coordinate sequence the rerouting requires minimum additional traffic (or minimum number of communication links which are used to deliver the broadcasted message)

Simulation Model

In order to evaluate the algorithm a software model was written in REXX, a high-level language similar to Pascal, and available under the VM/SP™ and OS/2™ operating systems. The model was divided into three modules. The first module (MAIN PROGRAM) dealt with fault generation, checking, and record keeping. The second module (BACKTRACE SUBROUTINE) dealt with the generation of the Path Coordinate Sequence, backtracing through that sequence, and updating the tree to reflect new paths to those nodes already in existence. The third module (GENTREE SUBROUTINE) handled the generation of new links to nodes deleted due to faults, as well as the updating of the tree to reflect the modifications.

Main Program Module

The main program generates random faults, checks to see if the faults are present in the tree, and collects resultant data (faults detected, number of faults applied, etc.). The inputs to this module are two: the dimension of the hypercube, and the number of faults that could be present in the system at the same time. For example, in the case of a 3-dimensional hypercube with number of faults equal to two, the main module will generate two links faults at random.

In order to simplify the program, it was decided to consider one fault at a time. For instance, if "down_links" = {1.3 0.4} indicating faults on the links connecting nodes 1-3 and nodes 0-4, the program will correct the fault 1.3 first. Correcting a fault means selecting the PCS, bactracking , modifying the tree for the existing nodes, and calling GENTREE. Once the first fault has been dealt with, then the two faults will be presented to the new tree. The module is best described by using pseudocode, as shown in Figure 6.

```
GET Dimension and "f" = number_of_faults
INITIALIZE tree
DO UNTIL "stop_signal" = 1
  GENERATE "down_links"
  DO UNTIL all members of "down_links"
           have been applied OR Loop_count
           is less than (2**"dimension")

    GET fault = nth_member of "down_links"
    IF fault is present in tree
      THEN
        CALL BACKTRACE with "fault" as
             parameter
        IF returns from BACKTRACE = error
          THEN
            EXIT with BACKTRACE error
          ELSE
            ITERATE
        END IF
    END IF
  END DO
  IF exit with error
    THEN
      INCREMENT appropriate error counters
      INCREMENT attempts counter
    ELSE
      INCREMENT attempts counter
  END IF
END DO
OUTPUT statistics
```

Figure 6. Pseudocode for the Main Program Module

Backtrace Module

The BACKTRACE module selects the PCS based on the "fault" generated in the MAIN Program. It also performs the backtracking for the PCS, and the modifications to the tree to reflect the new path to those nodes in existence. This becomes clear by following the pseudocode for this module, described in Figure 7.

Gentree Module

The GENTREE module receives the value of the last node reached by the PCS, and the value of "t". It will then proceed to generate new links based on the rules previously described in the Algorithm Description section. The pseudocode for this module is in Figure 8.

```
GET "fault"
PRUNE tree based on "fault"
GENERATE PCS
IF PCS generates parallel path to
   existing node
  THEN
    RETURN with BACKTRACE error
  ELSE
    MODIFY tree to reflect PCS
    CALL GENTREE with last node reached
         by the PCS and "t"
    IF GENTREE returns with error
      THEN
        RETURN with GENTREE error
      ELSE
        RETURN with no error
    END IF
END IF


Note: In the case of multiple values of "s"
      BACKTRACE will select a random "s"
      value from the given set.  Note that
      s = d only if no other choice of "s"
      is available.  This note applies to
      the third statement of the module
      'GENERATE PCS'.
```

Figure 7. Pseudocode for the Backtrace Module

Results/Analysis

Table 1 shows the coverage resulting when we apply multiple faults to a 3-, 4-, 5-, and 6-dimensional hypercube.

As we can see from the table, coverage falls below the 100% any time we have multiple faults. Failures of the algorithm were classified into four categories: loops, incomplete trees, duplicate nodes generated by BACKTRACE, and duplicate nodes generated by GENTREE.

Loops are generated when the algorithm is unable to resolve a set of faults. For example if we have a 3-dimensional hypercube with "down_links" = {1.3, 3.7}. The algorithm will handle fault 1.3 first, with a PCS = {2,1}. The tree generated is shown in Figure 5. Executing the algorithm, we then test the new tree to see if any member of "down_links" is

```
GET "last_node" and t
IF t = Dimension - 1
  THEN
    Do nothing and RETURN
  ELSE
    DO WHILE t < Dimension - 1
      IF "last_node" link t+1 Does not exist
        THEN
          GENERATE "son_node" from "last_node"
          thru link t+1
          MODIFY tree
          CALL GENTREE with "son_node" and t+1
        ELSE
          RETURN with GENTREE error
      END IF
    END DO
END IF
```

Figure 8. Pseudocode for the Gentree Module

present. We find that fault 3.7 is present, so we apply the algorithm one more time. The PCS for this fault is {1,2,1}, and the resulting tree is shown in Figure 9.

Again we apply the two faults {1.3 3.7} to the tree, and we find that fault 1.3 has reappeared. If we handle fault 1.3 the algorithm will give the same PCS as before {2,1}, which will cause fault 3.7 to reappear, hence a LOOP condition. Note--That the algorithm does not specify which "s" value to select, for the model we choose "s" at random.

The second category of algorithm failures is incomplete trees. This is the case when the final tree does not contain all nodes. An example of this case is when a node is completely isolated (all links to that node are faulty). However, there are other examples where all links to a given node are not faulty, yet the resultant tree is incomplete. Such is the case if we consider a 3-dimensional cube with "down_links" = {0.1, 0.4}.

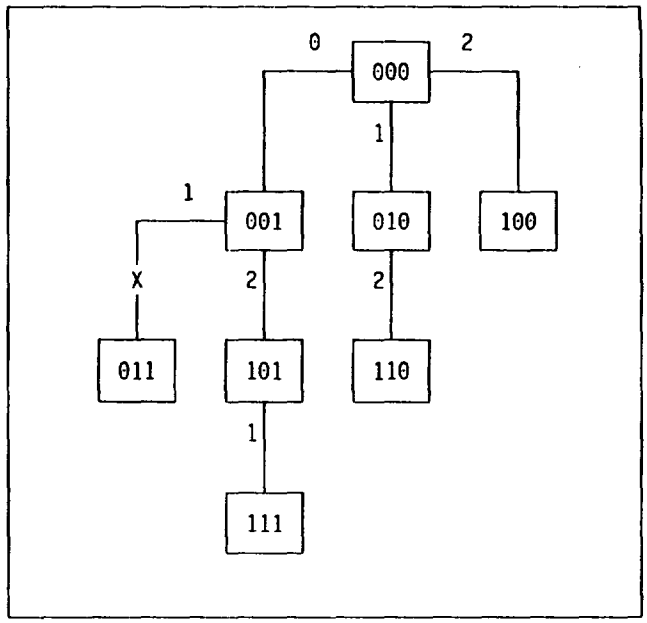

Figure 9. Example of tree where algorithm causes a loop.

Applying the algorithm to the first fault results in PCS = {2,0}, and the resulting tree is shown in Figure 10.

Applying the two faults {0.1 0.4} we find that 0.1 was handled correctly but 0.4 remains, so the algorithm is used once more. This time PCS = {1,2,1} with t = 2. The resulting tree is shown in Figure 11.

The next category of failure is BACKTRACE generating a duplicate node. An example of this condition occurs with faults in {0.4 2.6}. Fault 0.4 is handled first with PCS = {0,2,0}; that will put node 4 under node 5 as shown in Figure 12.

The second fault (2.6) is then handled, with PCS = {1,2,1}. The second member of the PCS regenerates a node that is already in existence (node 4). Following the PCS results in a link between node 0 and node 4, but node 4 could also be accessed via node 1 and

Table 1. Number of faults present and corresponding coverages

Faults	3-Dimensional	4-Dimensional	5-Dimensional	6-Dimensional
1	100%	100%	100%	100%
2	79.3%	94.3%	98.2%	99.3%
3	47.9%	83.2%	94.3%	98.0%
4	19.6%	68.0%	89.5%	96.6%

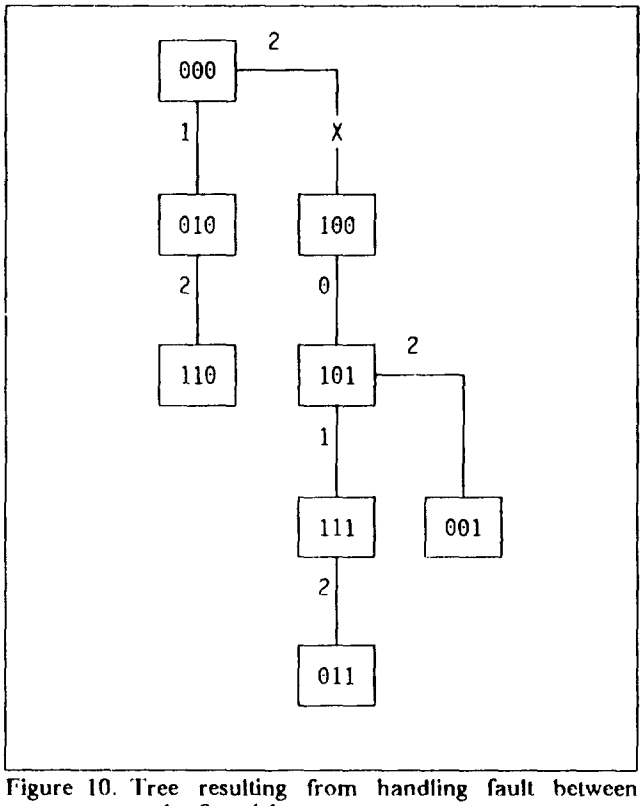

Figure 10. Tree resulting from handling fault between nodes 0 and 1

node 5. This violates the nonredundant clause of broadcast communication.

The last category of failures is GENTREE generating a duplicate node. This category is very similar to category 3 (BACKTRACE). An example of this condition is faults {0.2 2.3}. The problem is demonstrated by using the following PCS: {0,1,0} for the first fault (0.2), and {2,0} for the second fault (2.3). Again, node 4 gets duplicated this time under node 6.

The software model was modified in order to get an accurate breakdown of the types of failures. Table 2 shows the breakdown by failure types.

As we can see from the table, half of the algorithm failures were caused by GENTREE creating a duplicate node. This is directly related to the value of "t" used in the regeneration message. The other three types of algorithm failure are related more to the "s" value used. The case of Incomplete trees is the only one where failures could not be totally eliminated. This occurs when the number of faults is greater than or equal to the dimension of the hypercube, creating the possibility of a node being completely isolated.

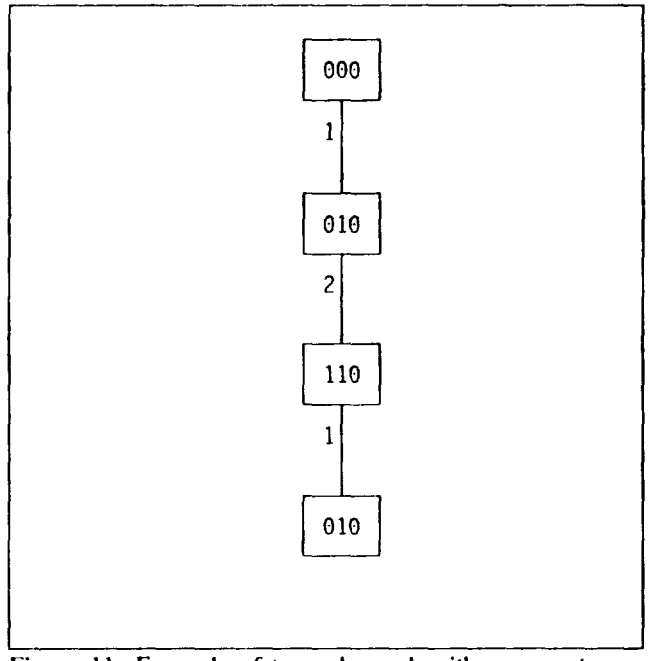

Figure 11. Example of tree where algorithm generates an incomplete tree.

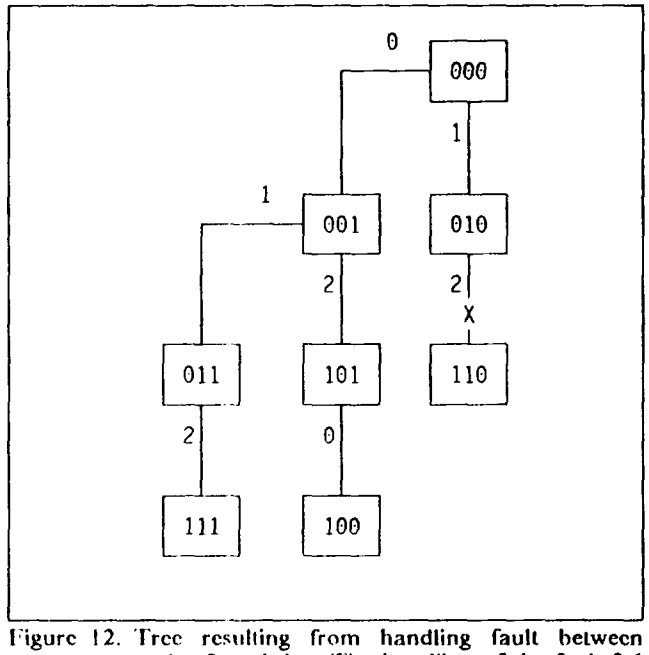

Figure 12. Tree resulting from handling fault between nodes 0 and 4. The handling of the fault 2.6 will result in node 4 being duplicated.

Table 2. Failure type breakdown.

Failure Types	3-Dimensional Cube			4-Dimensional Cube		
	2 Faults	3 Faults	4 Faults	2 Faults	3 Faults	4 Faults
Incomplete tree	26.5%	24.5%	25.7%	27.6%	26.5%	27.2%
Loops in tree	15.3%	17.1%	16.4%	13.1%	15.5%	15.4%
Backtrace Dup	4.1%	8.0%	8.2%	7.4%	6.9%	5.7%
Gentree Dup	54.2%	50.4%	49.7%	51.9%	51.1%	51.6%

Summary

In this paper we have evaluated Wu and Fernandez' fault-tolerant broadcasting algorithm on hypercube multicomputers. We have shown that the algorithm behaves as expected, with 100% coverage and minimum regeneration traffic, when one fault is present. When multiple faults are present coverage falls below 100%. As expected there is a relationship between the dimension of the hypercube and the algorithm coverage. The larger the dimension the higher the coverage for multiple faults. For instance, a 6-dimensional cube with four faults still gets coverage of greater than 96%. If we then consider the fact that the probability of a system having more than one fault present is very small, we can say that Wu and Fernandez' algorithm provides excellent coverage, while still keeping regeneration traffic to a minimum.

Analysis of the results indicates that the areas of tree regeneration and selection of "s" to generate the PCS are the most promising as far as improvements to the algorithm is concerned.

References

1. Reed, D. A. and R. M. Fujimoto, ***Multicomputer Networks: Message-Based Parallel Processing*** , The MIT Press, 1987.
2. Lan, Y. A., H. Esfahanian, and L. M. Ni, "Multicast in Hypercube Multiprocessors," ***Journal of Parallel and Distributed Computing 8*** , Jan 1, 1990, Pgs. 30-41.
3. Wu, J. and Eduardo B. Fernandez, "A Fault Tolerant Distributed Broadcast Algorithm for Cube-Connected Cycles," ***Proc. of 1990 Canadian Conference on Electrical and Computer Engineering***, Sept. 1990, Pgs. 51.4.1-51.4.4.
4. Brown, M. R., "Implementation and Analysis of Binomial Queue Algorithm," ***SIAM J. Computing 7*** , Aug 1978, Pgs. 298-319.
5. Saad. Y. and Schultz M. H., "Topological Properties of Hypercubes," ***IEEE Transactions on Computers***, Vol 37, No. 7, July 1988, Pgs. 867-872.
6. Banerjee, P., "Strategies for Reconfiguring Hypercubes Under Faults," ***Proc. of the 20th International Symposium on Fault-Tolerant Computing***, 1990, Pgs. 210-217.
7. Al-Dheloan, A. and B. Bose, "Efficient Fault-Tolerant Broadcasting Algorithm for the Hypercube," ***Proc. of the 4th Conference on Hypercubes Concurrent Computing and Applications*** , 1989, Pgs. 123-128.
8. Ramanathan, P. and K. G. Shin, "Reliable Broadcast in Hypercube Multicomputers," ***IEEE Transactions on Computers***, Vol 37, No. 12, Dec. 1988, Pgs. 1554-1657.

Trademark Acknoledgements -- VM/SP and OS/2 are Trademarks of IBM Corp.