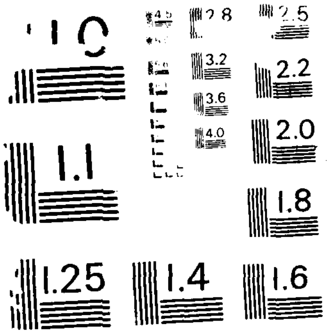
AD-A192 929 DISTRIBUTED FAULT TOLERANT EMBEDDINGS OF BINARY TREES 1/1
IN HYPERCUBES(U) PITTSBURGH UNIV PA DEPT OF COMPUTER
SCIENCE F PROVOST ET AL. TR-88-3

UNCLASSIFIED F/G 12/5 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

(4)

DEPARTMENT OF COMPUTER SCIENCE

Technical report 88-3

Distributed Fault Tolerant Embeddings of Binary Trees in Hypercubes*)

Foster Provost
and
Rami Melhem

University of Pittsburgh
Pittsburgh, Pennsylvania 15260

88 3 21 127

4

Technical report 88-3

Distributed Fault Tolerant Embeddings of Binary Trees in Hypercubes*)

Foster Provost
and
Rami Melhem

Department of Computer Science
The University of Pittsburgh

1988

*) This research is, in part, supported under ONR Contract N00014 80 C 04555.

Distributed Fault Tolerant Embedding of Binary Trees in Hypercubes[*)]

Foster J. Provost

Rami Melhem

Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260

ABSTRACT

In this paper we present a distributed algorithm for embedding binary trees in hypercubes. Starting with the root (invoked in some cube node by a host), each node is responsible for determining the addresses of its children, and for invoking the embedding algorithm for the subtree rooted at each child in the proper cube node. This distributed embedding, along with the wealth of communication links in the hypercube, leads to a high potential for fault tolerance. We demonstrate the fault tolerance capability by introducing tree restructuring techniques which may be used to tolerate faults during the initial embedding, as well as to remap nodes that fail at run-time. The distributed nature of the embedding eliminates the need for global knowledge of faulty nodes: each node must only know the status of its neighbors.

1. Introduction

Several researchers have studied the embedding of balanced binary trees in hypercubes. In doing so it is important to discover embeddings which preserve the adjacency of nodes so that communication can take place efficiently. It has been proven that an n-tree cannot be embedded in an n-cube with adjacency preserved.[8] Different methods have been suggested to bypass this difficulty. Bhatt and Ipsen [1] and Deshpande and Jenevein[2] embed a "two-rooted" n-tree in an n-cube, preserving adjacency; they use the extra root as a communications processor which routes information from the root to one of its logical children. This provides communication with a maximum of one "hop" between processors, where a hop denotes the passage of a message through a processor node, en route to another node. Johnsson[4] and Wu[8] take a different approach; they embed an n-tree in an $(n+1)$-cube preserving adjacency. Wu gives a recursively defined, bottom-up algorithm for determining a proper embedding; Johnsson, in contrast, gives a top-down algorithm. Both algorithms rely on a "bird's eye" picture of the entire embedding -- Wu's constantly rotates entire subtrees in order to achieve a proper alignment, while Johnsson's plucks information from different parts of the tree to determine the addresses on the next level of the embedding. Ipsen and Bhatt's algorithm [1] is essentially an extension of Wu's. In either case, the embedding has to be determined in advance (probably by a host) and tree nodes are statically assigned to processors.

The question of fault tolerance arises naturally at this point. If a tree node processor or communication link is faulty, the entire embedding has to be recalculated and tree nodes reassigned to processors. Even worse, if the fault occurs during processing, the entire structure has to be remapped before the processing can continue. This is time consuming and costly, especially if each tree node has to transport its program and data to the node's new location in the hypercube. In a time critical application, this expense would not be feasible.

Fault tolerance in tree connected multiprocessors has been considered by many researchers (see for example [3] and [6]). However in the previous research, fault tolerance has been achieved by adding redundant hardware to a tree structure such that the tree connection is preserved even in the presence of faults. Our approach, on the other hand, uses a hypercube-configured multiprocessor and takes advantage of the inherent redundancy of communication paths to remap faulty processors and avoid faulty links, thus producing a fault tolerant embedding of a binary tree structure. This approach is similar to that taken by Sami and Stefanelli [7] and Kung[5] for reconfiguration in meshes.

We present, in Section 2, an algorithm for embedding a binary n-tree in an (n+1)-cube which is entirely distributed and has a high potential for fault tolerance. The correctness of the algorithm is established by proving its equivalence to an instance of Wu's bottom-up algorithm. In Section 3 we show the fault tolerance capability of the distributed embedding by introducing a tree restructuring technique which may tolerate node or edge faults. The technique is extremely local and entirely distributed; in most cases only the failed node itself (or a node connected to a failed edge) is remapped, and only its parent and children are aware of the change. Any single fault can be tolerated as well as many multiple fault configurations. The tree is initially embedded with adjacency preserved; an embedding which bypasses a faulty processor (or a set of faulty processors) may contain one-hop communication between the remapped tree node and its logical neighbors. If the communications part of the processor is suitably designed, the one-hop degradation due to faulty processors will incur only a minor increase in communication time between nodes. The same applies if a faulty node may act as a switch (with trivial delay).

In Section 3.2, we enhance the algorithm so that it tolerates any configuration of two faults, and finally, in Section 4, we give simulation results which show the robustness of the fault tolerant algorithms.

2. A Top-Down, Distributed Embedding Algorithm

We have developed a top-down, distributed algorithm for embedding an $(n-1)$-tree in an n-cube, preserving adjacency. In this algorithm, each node receives a small packet of information from its parent, determines which nodes will be its children and sends them the information they need to continue the configuration. The information needed to determine the addresses of a node's children consists of: (i) the height of the subtree rooted at the node; and (ii) an "interchange word" which specifies some axes along which the subcube has to be reflected in order to yield a proper embedding.

The addresses of the node's children are calculated based on the bit positions which differ between the node's address and its child's. Since adjacent nodes in a hypercube differ in only a single bit position, a single integer will suffice to differentiate between the neighbors of a node. In the subsequent discussion, neighbors of a given node will be referred to by the bit position differentiating them from this node. The interchange word, which in our algorithm is passed from a parent to its children, consists of a string of bits. Numbering the bits with zero corresponding to the rightmost (least significant) bit, a "1" in the jth position specifies an interchange of the form (j/j+1), where an interchange of the form (i/k) indicates that subsequent children that are mapped to neighboring nodes across the ith dimension (if any) have to be remapped to the neighboring nodes across the kth dimension and vice versa.

More specifically, a node receives the height h of the subtree rooted there, and initially takes its left child to be its neighbor in the h dimension, and its right child the neighbor in the $(h-2)$ dimension. These bit positions are then modified by applying the interchanges specified in the node's interchange word. The word is read right to left, applying a given interchange to the bit position addresses of both children if the corresponding bit in the interchange word is set. To the right child, whose dimension has just been determined, the node sends height $(h-1)$ and its interchange word unmodified. To the left child the node sends height $(h-1)$ and its interchange word, modified by setting bit $(h-2)$. The embedding can be rooted at any node in the cube by invoking procedure EMBED there with the height h of the tree to be embedded and the interchange word rot =0.

```
procedure EMBED($h$ ,$rot$ )
begin
```

```
/* height $h$ of subtree to be rooted at this node */
/* interchange word $rot$ */

if $h=1$ then this is a leaf node

else if $h>1$ then

    $r:=h$:
    $l:=h-2$;

    for $i:=0$ to (length of $rot$) loop

    if (bit $i$ of $rot=1$) then

        if $(r=i)$ then $r:=i+1$;
        else if $(r=i+1)$ then $r:=i$;
        end if;

        if $(l=i)$ then $l:=i+1$;
        else if $(l=i+1)$ then $l:=i$;
        end if;

    end if;

    end loop;

end if;

$rot_r=rot$
$rot_l=rot+2^{h-2}$

invoke EMBED($h-1, rot_r$) at the node across dimension $r$   /* right child */

invoke EMBED($h-1, rot_l$) at the node across dimension $l$   /* left child */

end EMBED
```

We will now prove that this algorithm always yields a correct embedding.

2.1. Correctness of the Algorithm

In [8] Wu gives an algorithm for embedding an $(n-1)$-tree in an n-cube and shows that the embedding is correct. We will show that our algorithm always produces an embedding identical to an instance of Wu's and therefore is also always correct.

We first introduce a preliminary result. Let $R=x_{n-1}x_{n-2} \cdots x_0$ refer to a specific node and i,j refer to specific dimensions in the n-cube. For any node $Q=(y_{n-1} \cdots y_i \cdots y_j \cdots y_0)$ in the hypercube, define the address transformation $\mathbf{T}_1^{i\,j,R}$ as follows:

$$\mathbf{T}_1^{\mathrm{i,j,R}}(Q)=y_{n-1}\cdots y_i\cdots \overline{y_j}\cdots y_0 \text{ if } y_i=x_i \text{ and } y_j=x_j$$
$$\mathbf{T}_1^{\mathrm{i,j,R}}(Q)=y_{n-1}\cdots \overline{y_i}\cdots y_j\cdots y_0 \text{ if } y_i=x_i \text{ and } y_j=\overline{x_j}$$
$$\mathbf{T}_1^{\mathrm{i,j,R}}(Q)=y_{n-1}\cdots y_i\cdots \overline{y_j}\cdots y_0 \text{ if } y_i=\overline{x_i} \text{ and } y_j=\overline{x_j}$$
$$\mathbf{T}_1^{\mathrm{i,j,R}}(Q)=y_{n-1}\cdots \overline{y_i}\cdots y_j\cdots y_0 \text{ if } y_i=\overline{x_i} \text{ and } y_j=x_j\,.$$

Given an embedding **E** of an $(n-1)$-tree in an n-cube rooted at node R, $\mathbf{T}_1^{\mathrm{i,j,R}}$ creates a new embedding $\mathbf{E}_1$ by performing a Gray-code rotation of the embedded structure about dimensions i and j.

Another transformation, $\mathbf{T}_2^{\mathrm{i,j,R}}$, is defined such that if the embedding **E** is rooted at node R, then $\mathbf{T}_2^{\mathrm{i,j,R}}$ creates a new embedding $\mathbf{E}_2$ by remapping the root via $\mathbf{T}_1^{\mathrm{i,j,R}}$ and embedding the rest of the tree via the same dimensional structure as in the original tree with dimensions i and j swapped. More precisely, the transformation $\mathbf{T}_2^{\mathrm{i,j,R}}$ is defined inductively on the level of the tree as follows:

$$\mathbf{T}_2^{\mathrm{i,j,R}}(R) = \mathbf{T}_1^{\mathrm{i,j,R}}(R).$$

For each node Q which is a child of some node P in the tree:

if $Q=P$ XOR 2^d $d \neq i$ and $d \neq j$ then $\mathbf{T}_2^{\mathrm{i,j,R}}(Q) = \mathbf{T}_2^{\mathrm{i,j,R}}(P)$ XOR 2^d.
if $Q=P$ XOR 2^i then $\mathbf{T}_2^{\mathrm{i,j,R}}(Q) = \mathbf{T}_2^{\mathrm{i,j,R}}(P)$ XOR 2^j.
if $Q=P$ XOR 2^j then $\mathbf{T}_2^{\mathrm{i,j,R}}(Q) = \mathbf{T}_2^{\mathrm{i,j,R}}(P)$ XOR 2^i.

The notation $Q=P$ XOR 2^d indicates that Q is the neighbor of P across dimension d; the addresses of P and Q differ only in bit position d.

Lemma 1: Let **E** be an embedding of an $(n-1)$-tree in an n-cube rooted at node $R=x_{n-1}x_{n-2}\cdots x_0$, if $\mathbf{E}_1$ and $\mathbf{E}_2$ are the two embeddings created by applying $\mathbf{T}_1^{\mathrm{i,j,R}}$ and $\mathbf{T}_2^{\mathrm{i,j,R}}$ to the nodes of **E**, then $\mathbf{E}_1$ and $\mathbf{E}_2$ are identical.

Proof:

We can see that under either transformation the physical adjacency of logically adjacent nodes will remain -- due to the fact that the first is a Gray-code rotation and by construction in the second. By induction on the level of the tree we show that a tree node in **E** is mapped to the same cube node in $\mathbf{E}_1$ and $\mathbf{E}_2$. The root (the only level 1 node) is mapped to the same node in $\mathbf{E}_1$ and $\mathbf{E}_2$. Suppose each level n tree node occupies the same cube node in $\mathbf{E}_1$ and $\mathbf{E}_2$. In $\mathbf{E}$. let N be a level $n+1$ node which is adjacent to its parent N' (a level n node) across dimension d. We show that N is mapped to the same cube node in $\mathbf{E}_1$ and $\mathbf{E}_2$. In $\mathbf{E}_1$ and $\mathbf{E}_2$, node N is mapped to cube nodes $\mathbf{T}_1^{\mathrm{i,j,R}}(N)$, and $\mathbf{T}_2^{\mathrm{i,j,R}}(N)$ respectively (similarly for N'). Now we consider three different cases.

Case (i) $d \neq i$ and $d \neq j$:

$$\begin{aligned} N &= N' \text{ XOR } 2^d \\ \mathbf{T}_1^{\mathrm{i,j,R}}(N) &= \mathbf{T}_1^{\mathrm{i,j,R}}(N' \text{ XOR } 2^d) \\ &= \mathbf{T}_1^{\mathrm{i,j,R}}(N') \text{ XOR } 2^d \\ &= \mathbf{T}_2^{\mathrm{i,j,R}}(N') \text{ XOR } 2^d \\ &= \mathbf{T}_2^{\mathrm{i,j,R}}(N) \end{aligned}$$

since by the induction hypothesis we know that $\mathbf{T}_1^{\mathrm{i,j,R}}(N')=\mathbf{T}_2^{\mathrm{i,j,R}}(N')$.

Case (ii) $d = i$

Notice that for any node P:

$$\mathbf{T}_1^{\mathrm{i,j,R}}(P \text{ XOR } 2^i)=\mathbf{T}_1^{\mathrm{i,j,R}}(P) \text{ XOR } 2^j$$
$$\mathbf{T}_1^{\mathrm{i,j,R}}(P \text{ XOR } 2^j)=\mathbf{T}_1^{\mathrm{i,j,R}}(P) \text{ XOR } 2^i.$$

From this and the definition of $\mathbf{T}_2^{\mathrm{i,j,R}}$, it follows that if $N = N' \text{ XOR } 2^i$ then

$$\begin{aligned} \mathbf{T}_1^{\mathrm{i,j,R}}(N) &= \mathbf{T}_1^{\mathrm{i,j,R}}(N' \text{ XOR } 2^i) \\ &= \mathbf{T}_1^{\mathrm{i,j,R}}(N') \text{ XOR } 2^j \\ &= \mathbf{T}_2^{\mathrm{i,j,R}}(N') \text{ XOR } 2^j \\ &= \mathbf{T}_2^{\mathrm{i,j,R}}(N' \text{ XOR } 2^i) \\ &= \mathbf{T}_2^{\mathrm{i,j,R}}(N) \end{aligned}$$

Case (iii) $d = j$ -- similar to case (ii).

Q.E.D.

We now prove by induction on the size of the tree that our algorithm produces an embedding isomorphic to an instance of Wu's algorithm. For a height one tree the proof is trivial. Suppose that the algorithms produce identical embeddings for trees of height n if we start our algorithm at the root produced by Wu's. Wu constructs an $(n+1)$-tree embedded in an $(n+2)$-cube by:

(i) taking two identical n-trees embedded in $(n+1)$-cubes rooted at R, such that R has a free neighbor A (no tree node mapped to it) across some dimension x and A has a free neighbor B across some dimension y (this is referred to as the "free-free-neighbor" property);

(ii) transforming one embedding via an adjacency preserving transformation **T** such that $\mathbf{T}(R)=A$ and $\mathbf{T}(A)=B$ -- we will call this cube the second, and the remaining cube (identical before transformation) the first;

(iii) connecting identical cube nodes in the two $(n+1)$-cubes across a new dimension $n+2$ prefixing addresses in the first subcube with 0 and in the second with 1;

(iv) making node $0A$ the new root (with the left child across dimension x and the right child across dimension $n+2$).

Notice that now the root $0A$ has as its left subtree the first n-tree embedding, rooted at R and has as its right subtree the rotated (second) n-tree embedding, rooted at $1A=1\mathbf{T}(R)$. (Notice also that the free-free-neighbor property still holds with $0A$ connected across dimension y to $0B$ which is connected across dimension $n+2$ to $1B=1\mathbf{T}(A)$). If we study this recursive procedure which assigns new dimension $n+2$ as the tree grows from an n-tree to an $(n+1)$-tree, we see that the connection between A and B in an $(n+1)$-cube is always across dimension $n+1$ and the connection between R and A is always across dimension n. This means that the transformation **T** in (ii) will have to be such that given $R=(x_n \cdots x_0)$, $\mathbf{T}(R)=A=R$ XOR 2^n and $\mathbf{T}(R$ XOR $2^n)=\mathbf{T}(A)=B=A$ XOR 2^{n+1}. $\mathbf{T}_1^{n\,n+1,R}$ is a transformation which will satisfy these criteria.

Suppose our algorithm for embedding an $(n+1)$-tree in an $(n+2)$-cube starts at the root A and recursively embeds its left subtree, an n-tree rooted across dimension n at $R=A$ XOR 2^n, in the $(n+1)$-subcube not containing dimension $n+2$. It then embeds its right subtree across dimension $n+2$; this subtree is an embedding of an n-tree (in the remaining $(n+1)$-subcube) which is identical to the embedding of the left subtree except for a swap between dimensions n and $n+1$. Applying $\mathbf{T}_2^{n\,n+1,R}$ to each node in the embedding of the left subtree produces the embedding of the right subtree.

We can see that if Wu's algorithm is applied for embedding an $(n+1)$-tree in an $(n+2)$-cube, and our algorithm is applied for embedding an $(n+1)$-tree in an $(n+2)$-cube rooted at the same node A :

(1) the left subtree of each is a recursively embedded n-tree rooted at A XOR 2^n and by the induction hypothesis they are identical;

(2) the right subtree is rooted at A XOR 2^{n+2}. Consider the embeddings of the right subtrees in each case.

(a) The right subtree in Wu's embedding can be embedded by applying $T_1^{n\ n+1\ R}$ to the left embedding

(b) The right subtree in our embedding can be embedded by applying $T_2^{n\ n+1\ R}$ to the left embedding.

(c) By Lemma 1, these right subtrees are identical

Thus, our algorithm creates an embedding identical to that created by an instance of Wu's, if it starts from the root produced by Wu's algorithm. Wu shows that her algorithm always produces a proper embedding, therefore our algorithm always produces a proper embedding (by proper we mean that adjacency is preserved, and that the mapping is one-to-one)

3. Modifications for Fault Tolerance

We make few assumptions regarding the architecture of the hypercube machine. Each node must have processing as well as communications capabilities. It is desirable, although not necessary, that these be separated into a computation processor and a communications processor, with message routing considerably faster than processing. Thus the "one-hop" incurred by the remapping of a failed processor will not significantly delay the network's (otherwise-adjacent node) communications.

Our fault model takes into account both faulty processors and faulty communications links. A node is considered faulty if:

(i) The node itself does not function properly (i.e. the computation and/or the communication part fails

(ii) A communications link used in the embedded tree structure for communications with the node's parent does not function properly.

We assume that a node can detect faults in it's immediate neighbors.

Our fault tolerance scheme is based on the fact that in embedding an n-tree into an $(n+1)$-cube (using our scheme), the $(n+1)$-cube can be divided into two n-cubes, one of which contains three-fourths of the nodes, the other contains only one-fourth. This can be seen by studying the embedding algorithm. Specifically, bit zero of the exchange word is never used; it is set by the level 2 nodes, but the level one nodes (the leaves) do not use it. This implies that any embeddings across dimension zero come from the initial assignment of l or r. Moreover note that in EMBED, only l can ever be zero, and it will be zero for every level 2 node (and only for these nodes). Thus, dimension zero separates the $(n+1)$-cube into two subcubes, one of which contains one-half of the leaves of the tree (one fourth of the total tree nodes), the other contains the rest of the tree (three-fourths of the total tree nodes).

This separation of the tree nodes yields a structure which is amenable to a very local and distributed fault tolerance scheme. If we look at a "mirror image" of the tree structure across dimension zero, we can notice that the only nodes which do not have a free "image" are the height 2 nodes and their left children. Hence the image subcube may be used to remap faulty processors for any node at a level higher than 2. For the leaf nodes and their parents, a different strategy must be employed. In other words, the scheme is divided into two parts. One applies to the leaves and their parents; the other applies to the rest of the nodes.

The algorithm for restructuring the tree due to faulty processors or links can be applied both at the initial embedding (to bypass already faulty nodes) or during execution (to locally and distributedly remap failing nodes. without disturbing processing elsewhere in the tree). Any single fault is guaranteed to be tolerated. This guarantee. by itself. downplays the strength of the algorithm. The extreme "localness" of the restructuring along with the relatively empty image subcube allows the system to tolerate many faults simultaneously. including groups and strings of faults in the original tree.

For nodes of height greater than two. if a child is detected to be faulty. the node will route all information (for that child) to the child's image across dimension zero. This will incur a "one-hop" delay. Once in the image subcube. the embedding algorithm of Section 2 will be used to determine each node's children. After the initial node remapping. a node in the image subcube will always attempt to "bounce back" (i.e. return processing to the original subcube) before taking over processing duties. Thus. for a single fault. the faulty processor's image will take over the processing duties. determine its children (the images of the faulty node's children). and route information to them. These image children will "bounce back" to the original children -- meaning that they will act as "one-hop" communications elements between the remapped node and its children. The net effect is a distributed shift of the node into the image subtree with the rest of the tree structure remaining unchanged (see Figure 1). If one of the original children is also faulty. its image will not be able to bounce back and hence will automatically take over the processing: the logical tree structure in each branch will return to the original subcube as soon as a non-faulty node is available. In this way sections or branches of the original tree can be faulty. as long as their images are not (see Figure 3).

For nodes on the bottom two levels (leaves and their parents) a special remapping must be employed. The parent will bypass a faulty node by routing the information for one leaf through the other (see Figure 2). At this point we must note that for each height 2 node (leaves' parent). the left child is originally connected across dimension zero and the

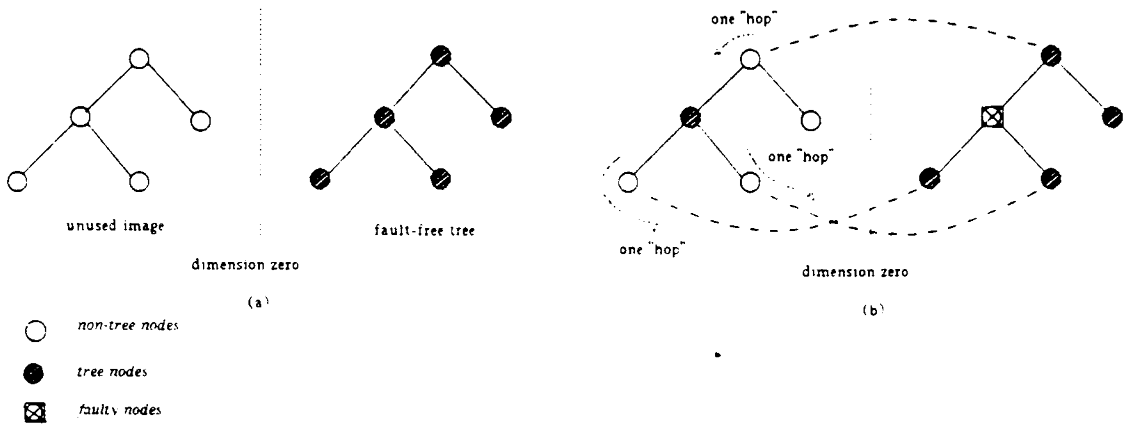


Figure 1
(a) a fault-free tree and its image
(b) local remapping for a single fault

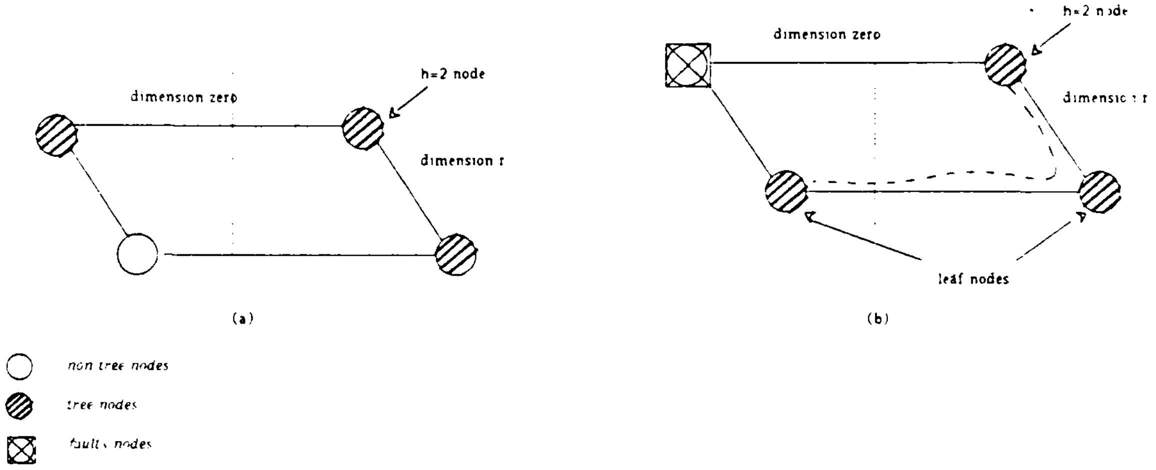


Figure 2
(a) fault-free leaf embedding
(b) remapping of a faulty leaf node

right child is originally connected across some other dimension (we refer to this dimension as dimension r). If a child of a height 2 node is faulty, the algorithm will map it to the node diagonally across dimensions r and zero. Whether the parent node is in the image subcube or the original subcube, its children occupy the same two cube nodes, mirrored about dimension zero. We consider the two-dimensional subcube containing the parent and its two children; if one of these four nodes is faulty, we will use the other three -- never incurring more than a one-hop delay in communications between logically adjacent nodes.

It can be seen that communications between any two logically adjacent nodes will incur at most a "one-hop" delay. This is the cost of the fault tolerant scheme. Notice also

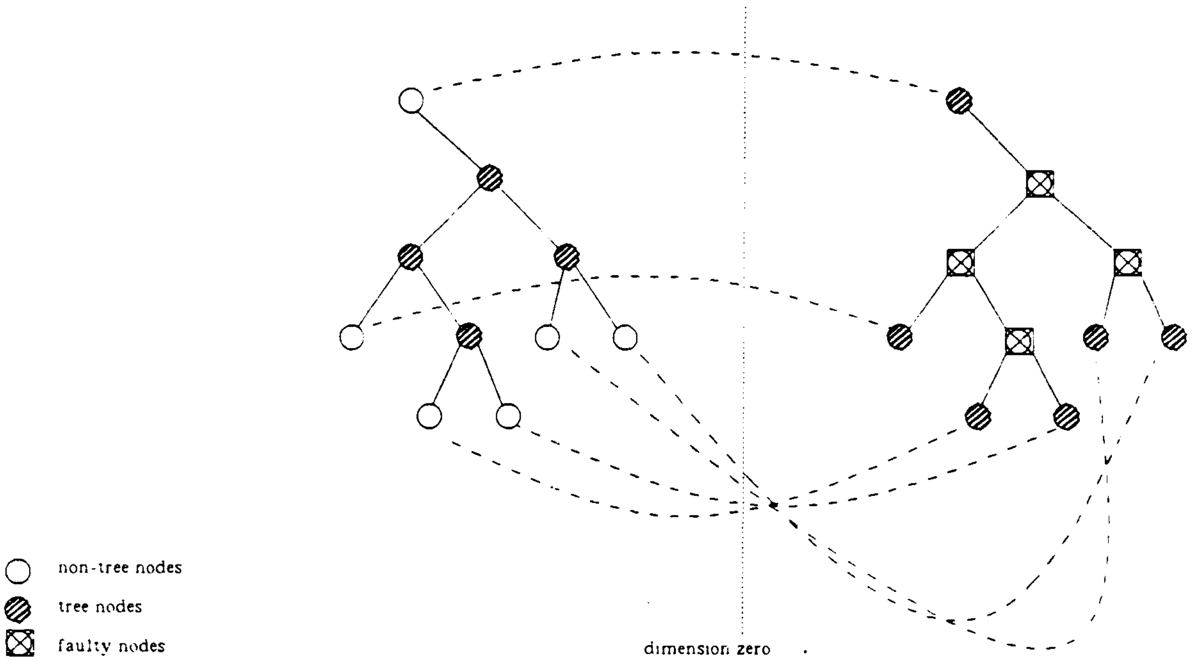


Figure 3 - remapping of a faulty section of the tree

that the remapping of an interior ($h > 2$) node will never interfere with the remapping of another node, and that no remapping of a node in one parent/leaf triad will interfere with the remapping of any node outside that triad.

3.1. The Fault Tolerant Algorithm

The crucial step is the remapping of nodes originally mapped to faulty processors. We assume that a node will remember the addresses of its parent and children for subsequent information routing. The remapping algorithm is very simple, consisting of only a few cases. The only technicality which has been added is a method for determining when to "bounce back." Different schemes can be envisioned, we use a pair of "correction" bits which signify different states in the remapping process:

00 -- indicates a node in the original embedding;

01 -- indicates the first in a sequence of remapped nodes in the image cube;

11 -- indicates any subsequent nodes which must decide whether to bounce back or to take over processing.

When we say "invoke" we refer to initiating the embedding of the appropriate subtree at the node specified. When we say " a node across dimension x/y" we designate a node whose address differs from the current node in both the x and y positions. In the single fault case, a fault-free path is guaranteed to exist to that node either through dimension x and then y or through y and then x.

```
procedure EMBED_FT($h$, $rot$, $correction$)
begin

    determine $r$ and $l$ and interchange words $rot_r$ and $rot_l$ according to EMBED

    if $h = 1$ then this is a leaf node

    else if $h > 2$ then

        case $correction$ of

            00 =>
              for $\alpha \in \{r, l\}$ loop
                  if node across dimension $\alpha$ is non-faulty then
                        invoke EMBED_FT($h-1$, $rot_\alpha$, 00) at the node across dimension $\alpha$
                  else
                        invoke EMBED_FT($h-1$, $rot_\alpha$, 01) at the node across dimension zero/$\alpha$
              end loop

            01 =>
                  invoke EMBED_FT($h-1$, $rot_r$, 11) at node across dimension $r$
                  invoke EMBED_FT($h-1$, $rot_l$, 11) at node across dimension $l$


            11 =>

                  if node across dimension zero is non-faulty then
                        invoke EMBED_FT($h$, $rot$, 00) at node across dimension zero
                  else
                        invoke EMBED_FT($h-1$, $rot_r$, 11) at node across dimension $r$
                        invoke EMBED_FT($h-1$, $rot_l$, 11) at node across dimension $l$

        end case;

    else if $h = 2$ then      /* $l$ = zero */

              for $\alpha \in \{r, \text{zero}\}$ loop
                  if node across dimension $\alpha$ is  non-faulty then
                        invoke EMBED_FT($h-1$, $rot_\alpha$, 00) at the node across dimension $\alpha$
                  else
                        invoke EMBED_FT($h-1$, $rot_\alpha$, 01) at the node across dimension zero/$r$
              end loop
    end if;
end EMBED_FT;
```

Even though the algorithm is presented such that it produces an initial fault tolerant embedding, it should be clear that the same strategy may be used to remap nodes that become faulty during runtime into non-faulty nodes. In fact, if we were to ignore runtime faults and consider only faults at the initial tree embedding, the algorithm EMBED_FT might be written in a simpler form. Namely, there would be no need to bounce back to the original subcube. For this we can disallow state 11 and replace it with state 00.

3.2. An Enhanced Fault Tolerant Algorithm

The algorithm EMBED_FT is very good for tolerating multiple faults, but certain double fault configurations cannot be tolerated. When a node is faulty, its image across dimension zero should not be faulty -- neither should the images of its parent or its children. Thus for each node that is faulty there are four nodes which must all be non-faulty. We now modify our algorithm so that it will tolerate any configuration of two faults, and many multiple fault configurations. For this, we consider distance 2 fault detection. This could be implemented by having a node request a status report from some distance 2 neighbor via both one-hop paths, if it fails to receive an acknowledgement, it assumes (for the purpose of the algorithm) that this neighbor is faulty. We can modify the scheme for immediate neighbor fault detection by adding an extra fault check state, but for clarity we assume distance 2 fault detection.

The modifications stem from the fact that if a node can communicate with another node across dimension x and then y (one-hop), it can also communicate across dimension y and then x (also one-hop). Thus if we encounter a second fault when attempting to remap a node, we can initiate the embedding at a neighbor and "one-hop" to the children bypassing the faulty node. We also use backtracking if the embedding gets stuck due to faults. If a node discovers that it cannot embed its children, it declares itself faulty and lets its parent worry about reconfiguration.

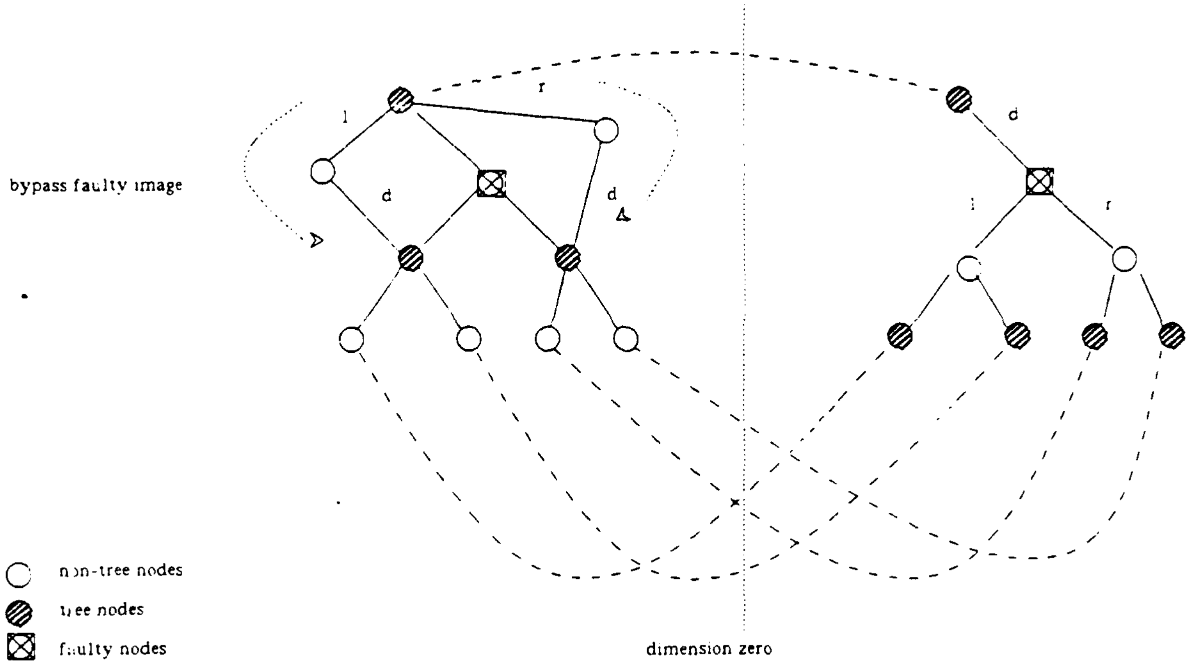


Figure 4 - a node and its image are both faulty

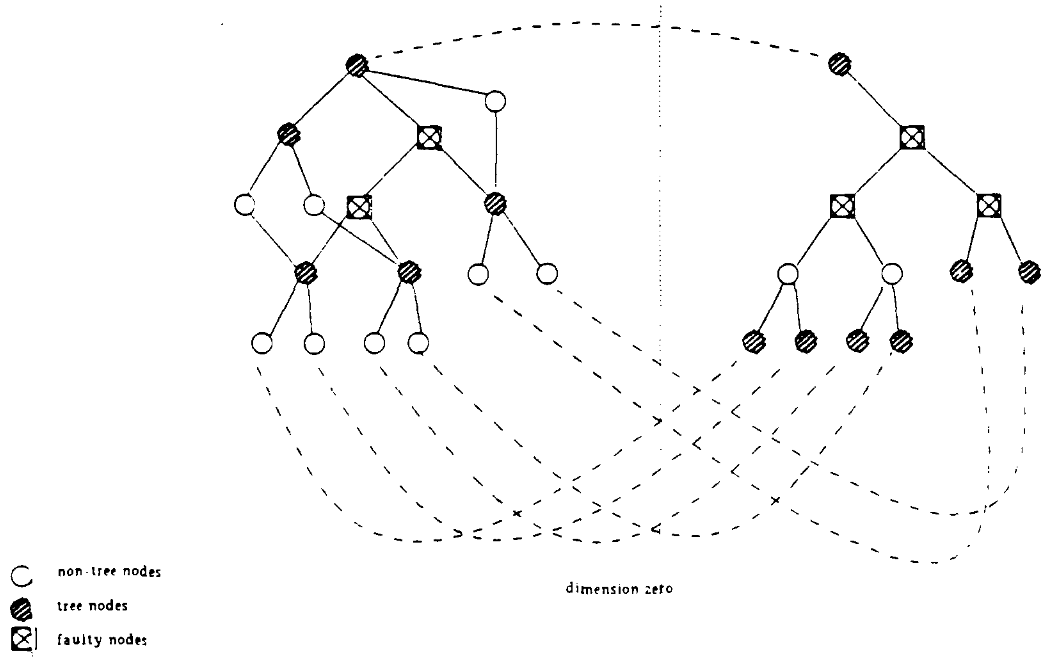


Figure 5 - a section of the tree and a section of its image are faulty

In our modified algorithm, a faulty node is remapped to the image of its parent across dimension zero. This node's children are then remapped to the images of the nodes where they would have been mapped in a fault-free embedding (see Figure 4). Thus, there are two one-hop paths from the remapped node to its children, and a second fault can be bypassed. For example, a node which was originally adjacent to its parent across dimension d (and whose children were across dimensions r and l) is remapped to its parent's neighbor across dimension zero (due to a fault in the original embedding). It can now map its children across dimensions d and then α (where $\alpha \in \{r, l\}$). If the node across dimension d is faulty, there is still a fault-free path across dimensions α and then d. This "bypass" of the node across dimension d enables the embedding algorithm to tolerate any configuration of two faults. As in the single fault algorithm, we indicate the traversal of dimensions x and y in either order (depending upon the position of faults) by "across dimensions x/y." In our example, if the node across dimensions d/α is faulty, the corresponding child can be mapped to the node across dimension α, and its children can be remapped using a similar bypass routing. In general, if a large subtree and its corresponding image are faulty, an embedding can still be realized with only one-hop communication; an example is given in figure 5.

The algorithm utilizes the relative emptiness of the image subcube to embed leaf nodes which, because of certain fault configurations, would otherwise not be successfully embedded. A leaf node's parent ($h=2$ node) in the image subcube that cannot embed the leaf in the node specified by EMBED attempt to embed the leaf in any unoccupied, fault-free node (at most distance 2) whose address's bit parity is different from that of a leaf node's. It can be shown that this will never interfere with a fault-free embedding of another part of the tree.

In the following algorithm, two correction bits are used to record the state of the remapping process as follows:

00 -- indicates a node in the original embedding;

10 -- indicates a node in the image subcube which must map its children using bypass routing;

01 -- indicates that the current node is mapped to the image of the node's position in a fault-free embedding;

11 -- indicates any subsequent node in the image subcube which must decide whether to bounce back or take over processing.

A handshaking protocol is necessary to enable the backtracking to function properly. When backtracking occurs, some nodes may have to be removed from the tree structure and should become available for subsequent embedding attempts. There are four different types of messages needed to synchronize the embedding. When a node decides to backtrack, an "invalidate" message is passed to any child at which the embedding of a subtree has already been invoked; this message is propagated throughout the subtree. Then a "failure" message is passed to the node's parent to indicate that the corresponding subtree could not be rooted at this node. Since the algorithm is distributed among the nodes, a "confirmation" message should be propagated from the leaf nodes to the root to indicate a successful embedding, and in response the root should initiate a "success" message which will propagate through the tree. This is a typical handshaking protocol used in distributed computing.

```
procedure EMBED_FT_2($h, rot, correction, d$)
begin

    if this node is already in use as a tree node then backtrack

    determine $r$ and $l$ and interchange words $rot_r$ and $rot_l$ according to EMBED

    if $h = 1$ then this is a leaf node

    else if $h > 2$ then

        case $correction$ of

            00 =>
              if any 2 of neighbors across dimensions $r, l$ or zero are faulty then backtrack
                  /* parent will reconfigure */
              end if
              for $\alpha \in \{r, l\}$ loop
                  if node across dimension $\alpha$ is non-faulty then
                        invoke EMBED_FT_2($h-1, rot_\alpha$, 00, 0) at the node across dimension $\alpha$
                  else
                        invoke EMBED_FT_2($h-1, rot_\alpha$, 10, $\alpha$) at the node across dimension zero
                  end if
```

```
                end loop

              10 =>
                for $\alpha \in \{r, l\}$ loop
                    if nodes across both dimensions $\alpha$ and $d$ are faulty then backtrack
                    if node across dimensions $\alpha/d$ is non-faulty then
                          invoke EMBED_FT_2($h-1, rot_\alpha$,01,0) across dimensions $\alpha/d$
                    else
                          invoke EMBED_FT_2($h-1, rot_\alpha$,10,$d$ ) across dimension $\alpha$
                    end if

              01 =>
                for $\alpha \in \{r, l\}$ loop
                    if node across dimensions $\alpha$/zero is non-faulty then
                          invoke EMBED_FT($h$ --1,$rot_\alpha$,00,0) at the node across dimensions $\alpha$/zero
                    else
                          invoke EMBED_FT($h-1, rot_\alpha$,11,0) at the node across dimension $\alpha$
                end loop


              11 =>
                    if neighbor across dimension zero is non-faulty then
                          invoke EMBED_FT_2($h$ ,$rot$ ,00,0) at node across dimension zero

                    else
                          invoke EMBED_FT_2($h-1, rot_r$ ,11,0) at node across dimension $r$
                          invoke EMBED_FT_2($h-1, rot_l$ ,11,0) at node across dimension $l$

          end case;

    else if $h$ =2 then      /* $l$ =zero */

          case correction of

              00 =>
                if any 2 of nodes across $r$ , zero or $r$ and zero are faulty then backtrack
                for $\alpha \in \{r$ ,zero$\}$ loop
                    if node across dimension $\alpha$ is non-faulty then
                          invoke EMBED_FT_2($h-1, rot_\alpha$,00,0) at the node across dimension $\alpha$
                    else
                          invoke EMBED_FT_2($h-1, rot_\alpha$,00,0) at the node across dimensions zero/$r$
                end loop

              01||11 =>
                for $\alpha \in \{r, l\}$ loop
                    if neighbor across dimension $\alpha$ is non-faulty then
                          invoke EMBED_FT_2(1,$rot_\alpha$,00,0) across dimension $\alpha$
                    else
                          invoke EMBED_FT_2(1,$rot_\alpha$,00,0)
                          at any unoccupied, non-faulty neighbor
                    end if

              10 =>
                for $\alpha \in \{r, l\}$ loop
```

```
                  if neighbor across dimensions $d / \alpha$ is non-faulty then
                        invoke EMBED_FT_2(1,$rot_o$,00,0) across dimensions $d / \alpha$
                  else
                        invoke EMBED_FT_2(1,$rot_o$,00,0)
                         at any unoccupied, non-faulty, distance 2 neighbor
                  end if

            end case

end EMBED_FT_2;
```

4. Performance Evaluation and Concluding Remarks

In order to test the robustness of our fault-tolerant algorithm, we performed a simulation study with various size hypercubes and varying numbers of faults. We always invoked the algorithm at a non-faulty node, and simulated the fault-tolerant embeddings on many different random fault configurations.

The simulation programs were written in C and the random faults were generated by calls to the Unix system function srandom(). Attempts were made to embed a tree of a given height for different distributions of a specific number of faults, and results were collected as to the number of successful attempts. For the first fault tolerant algorithm, EMBED_FT, the simulation results showed, as expected, that embeddings with zero or one fault were always successful (100 percent of the attempts succeeded). For the enhanced algorithm, EMBED_FT_2, embeddings with up to two faults were always successful. The performance of the embedding algorithms degraded nicely as the number of faults increased. The results for trees of heights 6 and 8 are shown in Figures 6 and 7, respectively. From these figures it is clear that the additional complexity in EMBED_FT_2 provides a more robust fault tolerant embedding algorithm.

The above results show the average number of faults that can be tolerated given an initial non-faulty starting node. This corresponds to the case of run-time fault-tolerance. At run-time, the basic structure of the tree embedding has already been established and, as faults occur, this basic structure should not change. The problem of fault tolerance at the initial embedding of the tree in the hypercube is slightly different. If the algorithm fails to

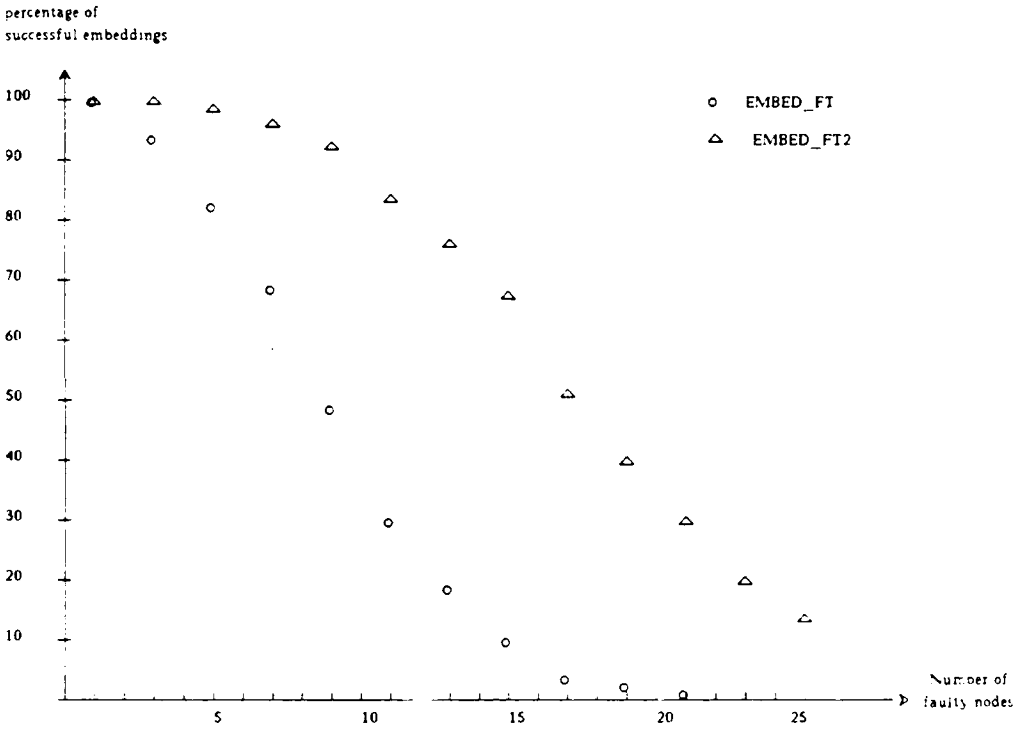


Figure 6 - Performance of the Fault Tolerant Algorithms
for a height 6 tree in a 128 node hypercube.

embed the tree starting at the given root, there is no reason why it might not succeed from another root. Also, if the algorithm fails to embed the tree for some set of faults, it is possible that choosing some dimension other than zero as the dimension across which faulty nodes are remapped may produce a successful embedding (this will, of course, change the basic structure of the embedding). We investigated the first of these two possibilities: specifically, if the algorithm EMBED_FT_2 failed to embed a tree rooted at a given node, alternative nodes were tried until either a successful embedding was achieved, or the algorithm failed to embed a tree rooted at any of the cube's nodes. Tables 1 and 2 give the results for trees of heights 6 and 8, respectively. For each specific number of faults, 100 different fault distributions were considered. The number of times that a successful embedding was achieved and the average number of roots tried before success are recorded in the second and third rows of the tables.

We tried to make as few assumptions as possible regarding the hardware model. The fault model includes both node and link failures and assumes that the

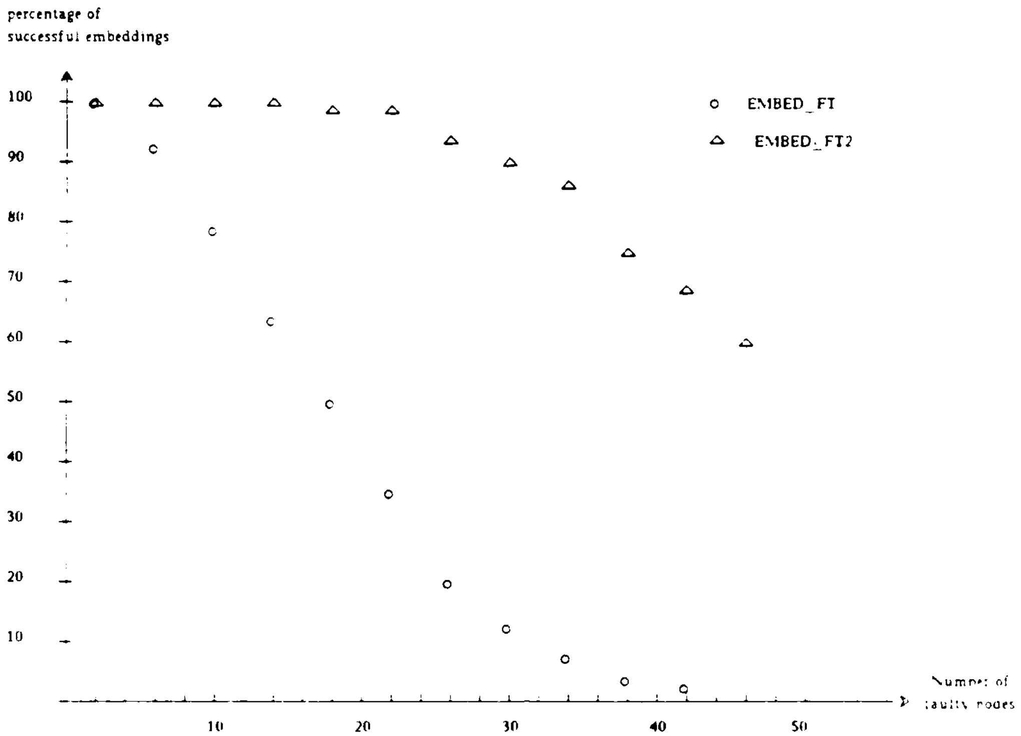


Figure 7 - Performance of the Fault Tolerant Algorithms
for a height 8 tree in a 512 node hypercube.

Number of faults	5	10	15	20	25	30	35
Probability of finding a successful embedding	1.0	1.0	1.0	1.0	1.0	.92	.44
Average number of roots tried before success	1.0	1.2	2.0	4.0	16	40	55

Table 1 - Embedding attempts at different roots for a height 6 tree.

Number of faults	10	20	30	40	50	60	70	80	90	100	110
Probability of finding a successful embedding	1.0	1.0	1.0	1.0	1.0	1.0	1.0	.96	.72	.50	
Average number of roots tried before success	1.0	1.0	1.1	1.1	1.5	2.5	4.2	12	31	105	213

Table 2 - Embedding attempts at different roots for a height 8 tree.

communications routing facilities associated with a node fail along with the processing facilities. Certain hardware models would simplify the fault tolerant algorithms and make them more robust. If the communications part of a node remains operable when the computation part fails (e.g. there is a separate routing part), this node would not have to be bypassed for communications purposes and more fault configurations could be tolerated If upon failure a node short circuits all communication across a certain dimension (zero), a technique can be developed where only the relocated node is aware of any change in the embedded structure -- the parent and children send messages to the original location and these messages are transmitted across the short circuit to the node's image.

Finally, we notice that the algorithms that have been developed for embedding a binary n-tree in and $(n+1)$-cube[4],[8] and our algorithm all produce isomorphic embeddings (in fact, it can be shown that Johnsson's and our algorithm are isomorphic and correspond to an instance of Wu's). We conjecture that the recursive embedding of an n-tree in an $(n+1)$-cube is unique.

References

1. S. Bhatt and I. Ipsen, "How to Embed Trees in Hypercubes," Research Report YALEU DCS RR-443, December 1985.

2. S. Deshpande and R. Jenevein, "Scalability of a Binary Tree on a Hypercube," *Proceedings of the ICPP*, pp. 661-668, 1986 .

3. S. Hosseini, J. Kuhl, and S. Reddy, "Distributed Fault Tolerance of Tree Structures," *IEEE Transactions on Computers*, vol. C-36, no. 11, pp. 1378-1382, November 1987.

4. S.L. Johnsson, "Communication Efficient Basic Linear Algebra Computations on Hypercube Architectures," *Journal of Parallel and Distributed Computing*, vol. 4, pp. 133-172, 1987

5. S.Y. Kung, C.W. Chang, and C.W. Jen, "Real-Time Configuration for Fault-Tolerant VLSI Array Processors," *Proceedings of Real-Time Systems Symposium*, pp. 46-54, December 1986.

6. C. Raghavendra, A. Avizienis, and M. Ercegovac, "Fault-Tolerance in Binary Tree Architectures," *IEEE Transactions on Computers*, vol. C-33, no. 6, pp. 566-572, June 1984.

7. M. Sami and R. Stefanelli, "Reconfigurable Architectures for VLSI Implementation," *Proc. Nat'l Computer Conf.*, pp. 565-577, May 1983.

8. A. Wu, "Embedding of Tree Networks into Hypercubes," *Journal of Parallel and Distributed Computing*, vol. 2, pp. 238-249, 1985.