EMBEDDING TRIPLE-MODULAR REDUNDANCY INTO A HYPERCUBE ARCHITECTURE*

Daniel L. Kiskis and Kang G. Shin

Real-Time Computing Laboratory
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, MI 48109-2122

Abstract

This paper describes an embedding of Triple Modular Redundancy (TMR) into a binary hypercube. The goal is to improve fault tolerance by masking any single-point faults. Each module of an application task is triplicated and executed in parallel on three nodes of a 2-dimensional subcube (Q_2) of the hypercube. Each of these nodes also executes a voter process. The remaining node is used for message passing only. All outputs from the triplicated modules are voted on, and the voting results are transmitted to the appropriate destination. Thus, all interunit messages are also triplicated.

We propose an embedding of TMR into a hypercube which can be implemented in a manner transparent to the application program. Subcubes are allocated so that the address space for the TMR units is also a hypercube. Hence, the subcube allocation and intermodule communication schemes are defined to be analogous to the schemes used in the nonredundant system. The embedded system is proven to mask all single-point faults.

*This work is supported in part by the NASA under the grants NAG-1-492 and NAG-1-296. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not reflect the view of NASA.

1 Introduction

The hypercube architecture is shown to be well suited for a large number of applications. Some of these applications come from areas such as real-time computing where reliability is crucial. However, current commercially available hypercubes have little or no fault-tolerant capabilities. What is required is a means whereby the system can be made more reliable while still providing all the benefits of the hypercube topology.

The implementation of fault tolerant techniques on a hypercube architecture has not been investigated extensively. Two hardware approaches are notable. The first is a modification of the hypercube topology proposed by Rennels [8]. Spare nodes are added to the hypercube in such a way as to provide replacements for any node which may be determined to be faulty. The second is a hybrid system suggested by Harper [5]. This system would consist of clusters of the Fault Tolerant Parallel Processor (FTPP) connected to one another in a hypercube topology. Applicable software solutions, e.g., [10], tend to be those which are intended for any distributed architecture which possesses specified characteristics, not for a hypercube topology in particular.

Both of the hardware approaches preserve the hypercube topology. Rennels' system, however, assumes that a faulty processor can be identified. This is a strong assumption, and as such, the solution is incomplete. Harper's system is more robust; it can tolerate all types of faults. However, this degree of

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© ACM 1988 0-89791-278-0/88/0007/0337 $1.50

reliability comes at a great cost in hardware. This cost may be justified for critical applications, but it is prohibitive for less critical applications where the cost of reliability must be much lower.

The software solution mentioned above may be useful for some applications, but it is not designed specifically for the hypercube topology. Thus, it is not optimized for that topology, nor is it guaranteed to preserve the topology as seen by the application. While other research is being performed to provide fault-tolerant message routing or allocation of processors and subcubes in the presence of known faulty nodes, these efforts, although important, do not provide a comprehensive solution to the reliability problem.

In this paper we present initial results in the development of a comprehensive system for providing a reliable computing environment on a hypercube architecture. In particular, we show how TMR can be embedded into a hypercube so as to provide single fault masking while preserving the observed topology of the system. This paper is organized as follows. Section 2 presents the problem statement and assumptions. Section 3 gives a brief description of the areas to be discussed. Section 4 describes the form of Triple-Modular Redundancy (TMR) to be used. Section 5 shows how TMR is embedded into a hypercube and presents a proof that the embedding preserves the fault masking properties of TMR. We conclude with Section 6.

2 Problem Statement

Our goal is to describe an embedding of TMR into a hypercube architecture. The embedding should be transparent to the application program, i.e., it preserves the perceived topology of the system. The embedding must also preserve the fault masking properties of TMR. It must mask all single-point faults, except for malicious/Byzantine faults. Using an appropriate representation scheme we show that the embedding meets these criteria.

TMR was chosen for two reasons. First, it provides fault tolerance via spatial redundancy. Although spatial redundancy increases the hardware overhead, it reduces the time overhead. Reliability techniques such as checkpointing with rollback or temporal redundancy require less hardware but more time. In many systems requiring reliability, such as real-time systems, time is also critical. Thus, a fault-tolerance technique which reduces the time overhead is preferred. Second, it is a simple means of providing fault masking at the task level and it has well-known properties which can be exploited.

We will use the following assumptions.

A1: Only nonmalicious operational faults are considered.

A2: Faults are independent random events.

A3: Messages between processors always travel along a minimum length path in the hypercube if such a path exists.

A4: Errors generated by faults in the communication subsystem of the hypercube can be detected.

Operational faults are those which occur after the system has been put into service; faults in the system's design are not considered. A1 serves to restrict the fault class to those faults which can be masked by TMR techniques.

Given that the probability of a single component failing is low, by A2 the probability of more than one component failing at or near the same time is negligible. Thus, it can be further assumed that once a fault has occurred, no other fault will occur until recovery from the first fault is completed.

A3 provides minimum length paths for messages. Unless otherwise specified, we require that the path be defined as follows. Compare the binary addresses of the source and destination nodes. Beginning with either the high order or the low order bit of the source address, successively change each differing bit in the address until the destination address is obtained. The addresses generated at each step determine the path [9]. We will refer to the two possible routings as *high-to-low* and *low-to-high*, respectively. This routing scheme is required in section 5.2 to insure that all single-point faults are masked.

A4 assumes that an aknowledgement based protocol is present which utilizes error detecting codes. Such mechanisms are sufficient for detecting nonmalicious faults in the communication medium.

3 Overview

Application programs consist of *tasks* which can be executed on separate node processors. Tasks are independent in that they may be executed in parallel. However, data and control dependencies may exist between tasks. These dependencies can be represented using an undirected *task graph* where nodes and edges represent tasks and intertask communications, respectively. It is at the task level of granularity that we define the degree of fault tolerance of the system. The only faults considered will be those which affect the output of the task. Such faults can manifest themselves in two ways: incorrect data and no data [3]. This classification encompasses faults in processors which produce data and faults in processors or links which transmit data.

The problem is to triplicate a task and assign it to three processors. Output from the different instantiations of the task will be voted on by a triplicated voter. All output from the tasks is voted. This includes data, control, and I/O request messages. The combination of the triplicated task and the triplicated voter will henceforth be referred to as the *TMR unit.*

The hypercube topology has the property that a hypercube of a given size can be subdivided into some number of disjoint subcubes of smaller dimensions. In order to preserve and take advantage of this property, it is desired to embed the TMR unit into some subcube of a given size. The problem is to determine the smallest subcube into which the TMR unit can be embedded without compromising its fault masking capabilities.

4 Topology of the TMR unit

The TMR unit will consist of the three replicated versions of the task (which will be referred to as *modules*) and three *voters*. In some TMR systems only one voter is used, and that voter is assumed to be fault-free. Since our voters are to be implemented through software on general purpose hardware, this assumption cannot be made. A single voter would have the same probability of being faulty as a module. Hence, a single voter system would be no more fault-tolerant than a single nonreplicated task.

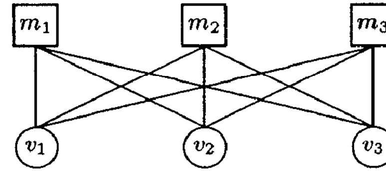

Figure 1: Graph of the TMR unit.

In order to provide masking of errors from a single fault, three voters are required. In a fault-free system, each voter receives the output from all three modules. The data are voted and the majority result is sent to the destination TMR unit. Although a faulty voter may provide erroneous data to another TMR unit, it will provide the data to only one module in that unit. The effects of that data will be masked by the voters of the module which received and then used the data.

The TMR unit can be represented by an undirected graph, as shown in Figure 1, where m_i are the modules and v_i are the voters, $i = 1, 2, 3$. Edges represent necessary communication paths between modules and voters.

We can describe the TMR unit using a representation scheme as defined in [6]. We will use the representation scheme $(\mathcal{S}, \mathcal{R}, \rho)$, where $\mathcal{S}$ is the specification class, $\mathcal{R}$ is the realization class, and $\rho : \mathcal{R} \rightarrow \mathcal{S}$ maps realizations to specifications. For a given $R \in \mathcal{R}$, if $\rho(R) = S \in \mathcal{S}$, then R *realizes* S. Since we are considering only operational faults, we can let $\mathcal{S} = \mathcal{R} = \{S | S$ is a network of sequential switching systems $\}$ and ρ is the identity function on R, that is, all realizations meet their specifications. In this discussion, we are considering $S =$ the TMR unit, and $R =$ an implementation of the TMR unit using dedicated hardware.

The TMR unit is a system with faults (S, F, ϕ) where $S \in \mathcal{S}$, F is a set of faults, and $\phi : F \rightarrow R$ is a function which maps a fault to the system resulting from the fault. $f \in F$ is *benign* if $\rho(\phi(f)) = S$.

For the TMR unit we define $F = \{0, 1\}^{15}$. $f \in F$ is a 15-tuple representing the 15 components in the

TMR unit component number	ident.	$\mathcal{F}$
1	m_1	p_2
2	m_2	p_3
3	m_3	p_1
4	v_1	p_2
5	v_2	p_3
6	v_3	p_1
7	$\overline{m_1v_1}$	p_2
8	$\overline{m_1v_2}$	$\overline{p_2p_3}$
9	$\overline{m_1v_3}$	$\overline{p_2p_0}, p_0, \overline{p_0p_1}$
10	$\overline{m_2v_1}$	$\overline{p_2p_3}$
11	$\overline{m_2v_2}$	p_3
12	$\overline{m_2v_3}$	$\overline{p_1p_3}$
13	$\overline{m_3v_1}$	$\overline{p_0p_1}, p_0, \overline{p_0p_2}$
14	$\overline{m_3v_2}$	$\overline{p_1p_3}$
15	$\overline{m_3v_3}$	p_1

Table 1: TMR unit embedding function

TMR unit, where $f = (f_{15}, f_{14}, \ldots, f_1)$ such that for $i = 1, 2, \ldots, 15$,

$$f_i = \begin{cases} 0 & \text{if component } i \text{ is fault-free} \\ 1 & \text{if component } i \text{ is faulty.} \end{cases}$$

The numbering of the components is given in Table 1. The embedding function $\mathcal{F}$ will be defined in Section 5.1; $\overline{m_iv_j}$ denotes the edge connecting module i and voter j.

In the following discussion, we will use X to denote a data value which is assumed to be erroneous or missing, therefore, its value is immaterial.

Definition 1 Given three data values x_1, x_2, and x_3, let

$$MAJ(x_1, x_2, x_3) = \begin{cases} x_i & \text{if } x_i = x_j, i \neq j, \\ & \quad i, j \in \{1, 2, 3\} \\ \text{X} & \text{if } x_1, x_2, \text{and } x_3 \text{ are} \\ & \quad \text{all distinct} \end{cases}$$

be the *majority function* on those data values.

Note that if $x_i = x_j = \text{X}$, for some $i \neq j$, then $\text{MAJ}(x_1, x_2, x_3) = \text{X}$.

Definition 2 Given a voter v, let *OUTPUT*(v) denote the *output* produced by v, where

$$OUTPUT(v_i) = \begin{cases} MAJ(m_1, m_2, m_3) & \text{if } v_i \text{ is fault-free} \\ \text{X} & \text{if } v_i \text{ is faulty} \end{cases}$$

$i = 1, 2, 3$. If either m_j or the link connecting m_j and v_i is faulty, then m_j =X.

Definition 3 We define $\beta(R) = (OUTPUT(v_1), OUTPUT(v_2), OUTPUT(v_3))$ to be the *behavior* of $R \in \mathcal{R}$, where R is the TMR unit.

Definition 4 A *tolerance relation* is a mapping $\tau : R \rightarrow S$ such that $R\tau S$, i.e., *R is within tolerance of S*, if a majority of the voters in R produce correct and equal outputs. This can alternately be stated as: $MAJ(OUTPUT(v_1), OUTPUT(v_2), OUTPUT(v_3)) \neq \text{X} \Rightarrow R\tau S$.

A fault $f \in F$ is τ-tolerated if $\phi(f)\tau S$. Given this definition, it can be seen that $\phi(f)\tau S$ if f is a single-point fault. Formally, a single-point fault is the case where, given $f \in F$, $f = (f_{15}, f_{14}, \ldots, f_1)$, $\sum_{i=1}^{15} f_i = 1$.

5 Embedding TMR

5.1 Embedding the TMR unit

In order to implement TMR on a hypercube, it is necessary to embed the TMR graph, G_{TMR}, into the i-dimensional hypercube graph, Q_i. This embedding is a function $\mathcal{F} : G_{TMR} \rightarrow Q_i$, which maps modules and voters to processors in the hypercube. Edges in the TMR unit map to individual links or paths in the hypercube.

In order to best preserve the properties of TMR, the function $\mathcal{F}$ should be one-to-one. The simplest such embedding is an isomorphic embedding. Modules and voters would be assigned to unique processors and edges would map to single links. In such a case, a single-point fault in the Q_i would correspond exactly to a single-point fault in the TMR unit. Hence, it is obvious that all single-point faults would be tolerated.

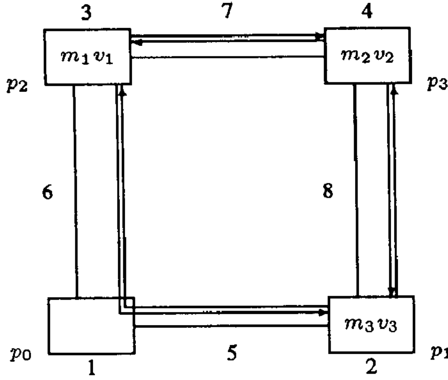

Figure 2: Embedding of TMR unit into Q_2.

It can be seen that G_{TMR} will not embed isomorphically into any Q_i. However, by allowing dilation of edges a homeomorphic embedding into a Q_4 can be found. Depending on the application, it may not be very cost-effective to require a Q_4 to implement the TMR unit. Such an embedding would reduce the computing power of a Q_n to that of a Q_{n-4}. We propose an embedding of G_{TMR} into a Q_2. Such an embedding is not one-to-one, but we will show that such a system will tolerate all single-point faults.

The embedding is shown in Figure 2. p_0 is not assigned modules or voters from this TMR unit. To each of the other three processors in the Q_2 is assigned a module and a voter. Each of these is marked with $m_i v_i$ to indicate which module and voter are assigned to it. The embedding of edges is shown by arrows from modules to voters. The embedding function $\mathcal{F}$ is given in Table 1.

In general, this embedding is static, but there is one exception. If a fault is detected in the path between p_1 and p_2, all subsequent messages between these processors will be sent via p_3. This exception is necessary for the proof of Theorem 3.

This embedded system can be described within the same $(\mathcal{S}, \mathcal{R}, \rho)$. Now, we are considering $R' \in \mathcal{R}$ where R' is the implementation of the TMR unit on the Q_2 and $\rho(R') = S =$ the TMR unit. The system with faults, (S, F', ϕ') is defined such that the fault set $F' = \{0,1\}^8$, where $f' \in F'$ is an 8-tuple such that $f' = (f'_8, f'_7, \ldots, f'_1)$ where

$$f'_i = \begin{cases} 0 & \text{if component } i \text{ is fault-free} \\ 1 & \text{if component } i \text{ is faulty.} \end{cases}$$

The numbering of the subscripts corresponds to the numbering of the components in Figure 2. For example, $f' = (0,0,0,1,0,0,0,0)$ is the case where $\overline{p_0 p_1}$ is faulty and all other components are fault-free. $\phi'(f')$ is the realization resulting from fault f'.

$\beta' = (OUTPUT(v_1), OUTPUT(v_2), OUTPUT(v_3))$ is defined exactly as β.

Our goal is to show that, given the presence of a single-point fault $f' \in F'$, $\phi'(f')\tau S$. That is, a fault in R' is τ-tolerated. In order to show this, we must demonstrate that any single-point fault in the Q_2 is τ-tolerated.

From Table 2 we can state the following theorem.

Theorem 1 If $f' \in F'$ such that f' is a single-point fault then $\phi'(f')\tau S$.

The success of this embedding relies on the fact that a TMR unit can actually mask a number of multiple faults. The embedding was selected such that single-point faults in the Q_2 correspond to tolerated multiple faults.

5.2 Embedding a Task Graph

5.2.1 TMR Version of the Task Graph

We have shown how a single task is triplicated to form a TMR unit, which can then be embedded into a Q_2. We can generalize this process to task graphs. Each task, t_i, in a task graph is replaced by a TMR unit, TMR$_i$. If an edge connects two tasks, t_i and t_j, in the task graph, then an edge connects each voter in TMR$_i$ with a unique module in TMR$_j$. An example is shown in Figure 3.

The problem which remains is to embed the TMR version of the task graph into the hypercube topology induced by the $\mathcal{F}$ embedding. We will first state some properties of the hypercube which we will need.

Fault $f' \in F'$	$\beta'(\phi'(f'))$
(0,0,0,0,0,0,0,1)	$(MAJ(m_1, m_2, X), MAJ(m_1, m_2, m_3), MAJ(X, m_2, m_3))$
(0,0,0,0,0,0,1,0)	$(MAJ(m_1, m_2, X), MAJ(m_1, m_2, X), X)$
(0,0,0,0,0,1,0,0)	$(X, MAJ(X, m_2, m_3), MAJ(X, m_2, m_3))$
(0,0,0,0,1,0,0,0)	$(MAJ(m_1, X, m_3), X, MAJ(m_1, X, m_3))$
(0,0,0,1,0,0,0,0)	$(MAJ(m_1, m_2, X), MAJ(m_1, m_2, m_3), MAJ(X, m_2, m_3))$
(0,0,1,0,0,0,0,0)	$(MAJ(m_1, m_2, X), MAJ(m_1, m_2, m_3), MAJ(X, m_2, m_3))$
(0,1,0,0,0,0,0,0)	$(MAJ(m_1, X, m_3), MAJ(X, m_2, m_3), MAJ(X, m_2, m_3))$
(1,0,0,0,0,0,0,0)	$(MAJ(m_1, m_2, m_3), MAJ(m_1, m_2, X), MAJ(m_1, X, m_3))$

Table 2: β' in the presence of single-point faults

5.2.2 Properties of the Hypercube Topology

It is well-known that the hypercube can be defined recursively as follows [4]:

1. Q_0 = the trivial graph consisting of one node
2. $Q_n = Q_{n-1} \odot K_2$

where $\odot$ is the Cartesian product of two graphs, and K_p is the complete graph with p nodes. Every node in an n-dimensional hypercube has associated with it an address. An address is a string $x \in \{0,1\}^n$. A subcube of the Q_n can be identified by an address string $s \in \{0,1,*\}^n$, where $*$ is the *don't care* symbol. For example, the address of the 2-dimensional subcube of a Q_4 formed by nodes 0010, 0011, 0110, and 0111 is 0*1*. The number of *'s in s is the dimension of the subcube [1].

Given this address scheme and the recursive definition of a hypercube, it follows that the Q_2's with addresses of the form $x**$, $x \in \{0,1\}^{n-2}$, form an $(n-2)$-dimensional hypercube with relation to each other. The Q_2's form the "nodes" of this $(n-2)$-dimensional hypercube. Each such "node" is connected to its neighbors by four edges.

In the $\mathcal{F}$ embedding, p_0 is only used for communication. Thus, one-fourth of the processors in each Q_2 are virtually idle. In order to utilize the computing power of that node, we make use of the following observation. Given a Q_2 with address $x**$, the address of p_0 for that Q_2 is $x00$. For $w \in \{0,1\}^{n-2}$, the set $\{y \mid y = w00\}$ defines a Q_{n-2}. Within that Q_{n-2} we have a set of Q_2's with addresses $z**00, z \in \{0,1\}^{n-4}$ which defines an $(n-4)$-dimensional hypercube into which we can embed TMR units as was done above.

If we apply this argument inductively we see that there are $n/2$ such *levels* of the hypercube, each one-fourth the size of the next larger one. The total number of usable Q_2's, i.e., the number of TMR units which can be supported, is thus

$$\sum_{k=0}^{\lfloor\frac{n-2}{2}\rfloor} 2^{2k} = \frac{2^n}{3} - \frac{1}{3},$$

if n is an even integer. The 1/3 in this equation is because the node 0^n is not actively used in any TMR unit except for message passing.

Hence, we have a series of nested hypercubes, which logically can be considered as either disjoint or connected, whichever is more convenient for a given application. The embedding of the TMR version of the task graph into the hypercube can now be defined. If the task graph will embed into a Q_{n-2}, then choose any level of sufficient dimension. Embed the graph in a manner analogous to the embedding of the simplex task graph. Except, TMR units will be embedded using the $\mathcal{F}$ embedding into Q_2's with addresses of the form $x**0^k$, where k is an even integer. Edges will be embedded into paths defined by either the high-to-low or the low-to-high routing strategy.

If the task graph will not embed into a Q_{n-2}, e.g., because it requires more than 2^{n-2} processors, then a new embedding must be found which will embed the task graph into multiple levels. As in the above, TMR units are embedded into Q_2's in their respective levels. Within each level, paths are embedded as

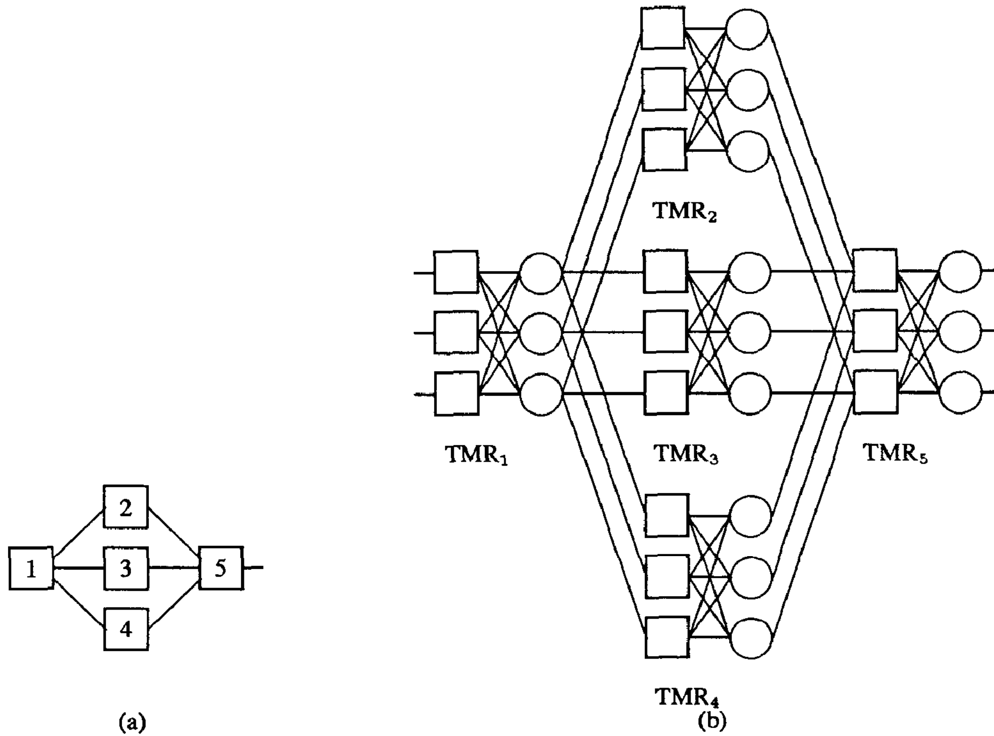

Figure 3: (a) nonredundant task graph (b) TMR version.

above. Paths between levels are defined as specified by the high-to-low strategy if going from a higher level to a lower level. That is, if going from a voter in a Q_2 with address $**0^k$ to a module in the Q_2 with address $y**0^l$ where $k < l$. When going from a lower level to a higher level, i.e., $k > l$, then the path defined by the low-to-high routing is used.

Theorem 2 Given three embedded TMR units TMR$_1$, TMR$_2$, and TMR$_3$ where TMR$_2$ and TMR$_3$ both send data to TMR$_1$, let voter $v_{2,\alpha}$ in TMR$_2$ send data to module $m_{1,\alpha}$ in TMR$_1$ and $v_{3,\gamma}$ in TMR$_3$ send data to $m_{1,\gamma}$ in TMR$_1$. If $\alpha \neq \gamma$ then the paths from $v_{2,\alpha}$ to $m_{1,\alpha}$ and from $v_{3,\gamma}$ to $m_{1,\gamma}$ are disjoint.

Proof: Let the addresses of TMR$_1$, TMR$_2$, and TMR$_3$ be x_1**0^k, x_2**0^l, and x_3**0^m, respectively, where k, l, and m are even non-negative integers. Without loss of generality, we let $v_{2,\alpha} = x_2 100^l, v_{3,\gamma} = x_3 110^m, m_{1,\alpha} = x_1 100^k$, and $m_{1,\gamma} = x_1 110^k$. The following three cases need to be considered.

Case 1: $k = l = m$. The TMR units are on the same level. Then, all nodes in the path from $v_{2,\alpha}$ to $m_{1,\alpha}$ have low order bits 100^l. Likewise, all nodes in the path from $v_{3,\gamma}$ to $m_{1,\gamma}$ have low order bits 110^m. Hence, the paths are disjoint.

Case 2: $l, m < k$ or $l, m > k$. Due to symmetry, these subcases are identical, so we will consider the case of $l, m > k$.

The paths we are considering have the following form:

$$x_2 100^l \rightarrow x_2 100^{l-k-2} 100^k \rightarrow x_1 100^k$$
$$x_3 110^m \rightarrow x_3 110^{m-k-2} 110^k \rightarrow x_1 110^k$$

where transitions (denoted $\rightarrow$) may involve multiple intermediate steps. In the source address, and during the first transition, the high order $n - k$ bits distinguish the paths. During the second transition, the low order $k + 2$ bits distinguish the paths. Thus, the paths are disjoint.

Case 3: $l > m = k, l < m = k, m > l = k$, or $m < l = k$. Again, because of symmetry these

subcases are identical. We will consider the case where $l > m = k$.

The paths in this case are of the form:

$$x_2 100^l \rightarrow x_2 100^{l-k-2} 100^k \rightarrow x_1 100^k$$
$$x_3 110^k \rightarrow x_1 110^k.$$

The high order $n - k$ bits distinguish all nodes generated in the first transition of $x_2 100^l$ to $x_2 100^{l-k-2} 100^k$ from the node $x_3 110^k$. During the next transition the low order $k + 2$ bits distinguish the nodes. Hence the paths are disjoint. ■

Corollary 1 Given two embedded TMR units, TMR_1 and TMR_2, such that TMR_1 sends data to TMR_2, then the paths between the voters in TMR_1 and the modules of TMR_2 are disjoint.

Proof: This is a special case of Theorem 2 where $TMR_2 = TMR_3$, i.e., $x_2 = x_3$ and $l = m$. ■

Theorem 3 The embedded TMR version of the task graph masks any single-point fault.

Proof: If no two TMR units are executing on the same Q_2, then by Theorems 1 and 2, the embedding preserves the structure and fault masking properties of the TMR version of the task graph. By the definition of TMR we know that all single-point faults will thus be masked.

If multiple TMR units are executing on the same Q_2, then a single-point fault acts as simultaneous multiple faults. A single-point fault can logically occur within a TMR unit, along a path between TMR units, or both. These cases will be considered separately.

Case 1: If two or more affected voters (executing on the same processor) send messages to the same TMR unit, the communication scheme guarantees that only one module in the destination TMR unit will be affected. Thus, the fault will be masked.

Case 2: The fault occurs along the path between two TMR units. In this case one module of the destination TMR unit will receive erroneous data and the voters of the TMR unit will mask the error. By Theorem 2, it is not possible for a single faulty component to affect the inputs to more than one module of a TMR unit. Thus, the errors generated by any such fault will be masked.

Case 3: In general, this case is subsumed by Cases 1 and 2. One special case needs to be considered. Consider a fault which occurs in a processor which is a p_0 for one Q_2 and a p_3 for another Q_2 at a different level, and the lower level Q_2 sends data to the higher level Q_2. In the higher level Q_2, such a fault will cause the module in p_3 to receive erroneous data. It will also cause the path between p_1 and p_2 to be faulty. Hence, none of the voters would produce correct output. However, we have assumed that a fault in a path can be detected, and in section 5.1 we showed how to reroute messages around this particular fault. Once this rerouting is performed, v_1 and v_3 will receive at least two correct outputs each and will then produce correct outputs. Therefore, this fault is also tolerated.

Therefore, all single-point faults are masked by the embedded TMR version of the task graph. ■

Note that this result is independent of the scheme used to embed the task graph. Hence, embedding the TMR version of the task graph is equivalent to embedding the nonredundant version of the task graph into a restricted address space. Thus, any embedding algorithm for the nonredundant embedding may be used.

This result can be generalized to having any number of TMR units (even the whole task graph) executed on a single Q_2. However, such a system would not take advantage of parallelism afforded by the distributed system. Thus, such an embedding would be an inefficient use of resources.

6 Conclusions

We have described a system for increasing the fault tolerance of a binary hypercube architecture system. For simplicity we have chosen TMR units as the building blocks of our system. We have shown how to embed a TMR unit and the corresponding TMR task graph into the hypercube. This embedded system was shown to tolerate all single-point faults.

Our goal was to improve the reliability of the system in a manner which was flexible, simple, and which was transparent to the application. The pro-

posed method meets this goal. The system is intended to be implemented in software, Thus, implementation will require a protocol for synchronization of data at the voters. Only loose synchronization implemented via the message passing mechanisms of the system is required. Synchronization will take place at all points where data is sent out of the TMR unit. The protocol is required to provide data synchronous voting. It must also be capable of detecting lost or out-of-sequence messages. Such a protocol is specified by Gunningberg [2] and verified to meet specifications by Gunningberg and Pehrson [3].

However, by implementing the system in software, we incur expensive overhead. The modules must sit idle as they await the receipt of acknowledgements from the voters. This wait is prolonged by communication delays and the execution times of the protocol and voting algorithm. Furthermore, due to the necessity of executing tasks on separate processors, implementation of the system reduces the usable computing power of the hypercube by a power of 4.

However, a software implementation allows greater control of the degree of reliability by the applications programmer. For example, an implementation of the TMR system may allow an application to choose whether it will run in simplex or TMR mode. Simplex mode may be chosen by less critical applications in order to better utilize resources. It can be seen that a system can support simplex and TMR modes simultaneously.

The next step in this research should be to model the system and analyze it. Desired characteristics to determine are correctness, performance, and reliability. It is especially important to determine the degree to which the performance of the system is degraded by the introduction of the specified fault tolerance techniques. Promising candidates for the modelling technique are the Timed Petri Net model or the Stochastic Activity Net model [7]. Such a models will allow the behavior of the protocol and voting algorithms to be modelled while providing timing information for performance analysis.

References

[1] M.-S. Chen and K. G. Shin. "Determination of a minimal subcube for interacting task modules". *Proceedings of the 2nd Conference on Hypercube Multiprocessors*, 122–129, 1987.

[2] P. Gunningberg. "Voting and redundancy management implemented by protocols in distributed systems". *Proceedings of the 13th International Symposium on Fault-Tolerant Computing*, 182–185, 1983.

[3] P. Gunningberg and B. Pehrson. "Specification and verification of a synchronization protocol for comparison of results". *Proceedings of the 15th International Symposium on Fault-Tolerant Computing*, 172–177, 1985.

[4] F. Harary, J. P. Hayes, and H.-J. Wu. *"A survey of the theory of cube graphs"*. Technical Report, Computing Research Laboratory - University of Michigan and Computing Research Laboratory - New Mexico State University, September 1986.

[5] R. E. Harper. *Critical issues in ultra-reliable parallel processing*. PhD thesis, Massachusetts Institute of Technology, June 1987.

[6] J. F. Meyer. "Reliable design of software". In R. Saeks and S. R. Liberty, editors, *Reliable Fault Analysis*, pages 112–123, Marcel Dekker, NY, 1977.

[7] J. F. Meyer, A. Movaghar, and W. H. Sanders. *"Stochastic activity networks: structure, behavior, and application"*. Technical Report, Industrial Technology Institute, December 1984. ITI 84-7.

[8] D. A. Rennels. "On implementing fault-tolerance in binary hypercubes". *Proceedings of the 16th International Symposium on Fault-Tolerant Computing*, 344–349, 1986.

[9] Y. Saad and M. H. Schultz. *Data communication in hypercubes*. Technical Report, Yale University, October 1985. YALEU/DCS/RR-428.

[10] N. Theuretzbacher. "'VOTRICS': voting triple modular computing system". *Proceedings of the 16th International Symposium on Fault-Tolerant Computing*, 144–150, 1986.