Fault–Tolerant Task Management and Load Re–distribution on Massively Parallel Hypercube Systems

Ishfaq Ahmad
School of Computer and Information Science
Northeast Parallel Architectures Center, Syracuse University, Syracuse, NY 13244
Arif Ghafoor
School of Electrical Engineering, Purdue University, West Lafayette, IN 47907

Abstract

In a massively parallel multicomputer system, consisting of hundreds or thousands of processors, it is very likely that some of its components (processors and links) will fail. If such a system is dedicated to time–critical applications with certain deadlines to meet, such failures can not be tolerated. We present a scheme for managing real–time task allocation and load re–distribution with fault–tolerance for hypercube systems. A set of processors, called fault–control processors (FCPs), can be used for keeping the duplicate copies of tasks and re–allocating tasks if the original processors of those tasks fail. Two–level task redundancy is used by grouping the FCPs as primary and secondary for each processor. The proposed scheme provides a high degree of fault–tolerance since each FCP itself is monitored by other FCPs. Assuming a failure–repair system environment, the performance of the proposed strategy has been evaluated and compared with a fault–free environment for 256–node and 512–node hypercubes, through simulation experiments. We also introduce a measure of goodness, *success probability*, which represents the probability of re–allocated tasks meeting their deadlines despite the failure of processors. It is shown that using the proposed scheme, a large percentage of the re–scheduled tasks can still meet their deadlines. The probability of a task being lost altogether, due to multiple failures, has been shown to be extremely low.

1 Introduction

One of major challenges faced by the designers of large–scale multicomputers is to incorporate fault–tolerance into these systems. With thousands of processors, it is likely for some of them to fail. Systems that can heal in the presence of failed processors rather than halting are vital for supporting real–time teraflops power [16]. In addition to failures, it is also very likely that computing resources become unavailable for a variety of other reasons such as some processors may have to be shut off or isolated from the rest of the system for maintenance purpose or due to some other reasons. Means must be provided to handle failures and unavailability of processors so that the operation of the rest of the system remains uninterrupted and the performance is not adversely affected.

In addition to reliability and fault–tolerance, time is considered to be a crucial resource to manage in critical application since the occurrence of events, such as the execution of tasks, must follow some timing constraints [7]. For real–time environments, the performance of multicomputers needs to be evaluated by successful execution of individual tasks within certain time period, rather than the average system behavior. Real–time environments are further classified as soft and hard. In a soft real–time environment, a task is considered lost if it does not meet its deadline [6] whereas in a hard real–time system, the failure to meet a deadline can be catastrophic [14]. Deadlines of tasks are met by using some dynamic task scheduling and load migration strategies [14], [15], [17].

Dynamic task scheduling and load balancing is an important problem for both real–time and non real–time systems and considerable research has been done in this regard [1], [2], [3], [11], [12], [15], [17]. The problem becomes more complex in the presence of processors failures [2], [3]. The design of a large–scale multicomputer system for real–time applications entails efficient means of resource sharing such as load balancing on the computing nodes, to meet timing constraints. Furthermore, there must be provisions to redistribute the unfinished computational load from the failed processors to the operational processors and to redirect new tasks arriving at the failed processors. These two requirements entail a dynamic load re–distribution strategy. The load re–distribution strategy needs to be carefully designed since the re–injection of computational load from the failed processors to the rest of the operational system can make some processors unstable.

In this paper, we propose a fault–tolerant load re–distribution strategy for large–scale multicomputers such as hypercube systems using a partitioning scheme. Partitioning of system with fault–tolerance is becoming more practical approach for large systems due to their massive sized, as has also been suggested for the new Connection Machine CM–5 [19]. The proposed scheme is general in the sense that it is applicable to any decentralized scheduling scheme. Also the proposed strategy is based upon a more realistic assumption about the failure situations, than the one given in earlier work [2]. In previous studies failure have been assumed to be static in that a given number of processors are assumed to fail simultaneously without any subsequent recovery or repair. However, the static failure assumption does not hold for real life systems where failures are generally randomly occurring events. Furthermore, the failed components can be diagnosed and repaired off–line and can be re–integrated back into the system [10]. The second realistic assumption made in this paper is that the incoming tasks to the failed processors cannot simply be ignored. Due to real–time con-

1063-9535/92 $3.00 © 1992 IEEE

straints, some means are needed to handle tasks arriving at the failed processors. In summary, the proposed strategy has the following objectives:

- The main objective is to propose a fault–tolerant task scheduling strategy with decentralized control which has the flexibility of allowing the running system to continue to be operational in spite of failures of processors. The failed processors are assumed to be repaired subsequently.
- In order to provide fault–tolerance for both real–time and non–real–time environments, a certain level of redundancy is required in case the processors with redundant copies also fail. In this paper, we assume two–level redundancy, that is, two redundant copies of every task are kept in the system.
- Re–scheduling of tasks should not cause instability in the system and tasks should be re–distributed to only those processors which are lightly loaded.
- For a real–time environment, re–scheduling of tasks should be quick and efficient so that most of the tasks, which would have been lost otherwise, can still meet their deadlines, without affecting the underlying normal decentralized load balancing algorithm.
- If some processors in the system are out of order, the total load entering into the system should not decrease. The newly arriving tasks at those processors should be redirected to the operational processors. This re–direction of new load should also not cause any instability and excessive performance degradation.

Load re–distribution with minimum impact on the normal load balancing can be achieved if the processors responsible for re–allocation of backup tasks, which we will refer to as Fault Control Processors (FCPs), have some partial knowledge of the global state of the system. The selection of FCPs is a crucial factor to the fault–tolerant performance of the system, especially in terms of re–scheduling and reducing chances of missing deadlines. In a centralized approach, a single FCP can manage redundant load for the whole system. However, the failure of the FCP itself can eliminate fault–tolerance capability, and therefore, this scheme is highly vulnerable to failure. On the other hand, if a fully distributed scheme is used where each node acts as an FCP, the global knowledge of the system load has to be acquired by each node which is a costly solution in terms of overhead; the generation of such overhead traffic can seriously affect the system's performance. If limited knowledge, such as only the load of neighboring processors, is used, the scope of re–allocation of a task to a suitable destination becomes rather limited. Clearly, a semi–distributed scheme with a fewer number of FCPs, each having some partial knowledge of the global state of the system, would be an alternative. Such a scheme is presented in this paper, which is based on partition of the hypercube topology into multiple symmetric regions (spheres). Each sphere is a cluster of processors and has a fault–tolerant control processor (FCP). Load re–distribution is carried out within individual spheres where the FCP of each sphere acts as a centralized controller for its own sphere. Each processor is assigned two types of FCPs for storing two redundant copies of each task present at the processor. FCPs with the first–level redundancy will be termed as *primary FCPs* and the other as *secondary FCPs.*

The performance of the proposed strategy has been evaluated and compared with no fault environment, through an extensive and detailed simulation. We show that the performance degradation due to failures is scalable to fault rate and does not depend critically on system parameters. Also, the probability of a task being lost altogether due to multiple failures has been shown to be extremely low.

2 System Partitioning and Assignment of FCPs

In this section, we describe the selection of FCPs and network partitioning for hypercube topologies. As shown in [1], partitioning of the interconnection network into multiple regions can be modeled as a problem that is NP–hard. Our solution employs combinatorial structure *Hadamard matrix,* for partitioning the network into multiple spheres such that each sphere contains an FCP at its center while each processor is assigned a set of FCPs which are responsible for the following tasks:
(a) maintaining redundant (backup) copies of the tasks in the network,
(b) monitoring failures in their spheres,
(c) in case of a failure, redistributing tasks of the failed processor in individual spheres, and
(d) maintaining the load status of the processors in the spheres.

For maintaining backups, the assignment of processors to FCPs is independent of the physical distances in the network, which results in a rather simple rule of assignment. On the other hand, for the above mentioned functionalities (b), (c) and (d), we use spheres centered at FCPs, as discussed in Section 2.2. In this paper, we do not address the issue of fault–diagnosis and failure identification. We assume the availability of some such scheme.

Let an n dimensional hypercube topology of a system be represented by an undirected graph, $Q_n = <U, E>$ where U is the set of nodes and E is the set of edges (communication links) joining the nodes. The nodes (processors) are numbered as 0, 1, 2, ..., (N–1) and each number is represented by a binary code. The length of binary code for n dimensional hypercube is n, which also represents the number of edges (links) incident on each node. *The Hamming Distance,* H_{xy}, between two binary codewords, $x=(x_1, x_2, .. x_n)$ and $y=(y_1, y_2, ...y_n)$ of some length n, is defined as $H_{xy} = | \{ i \mid x_i \neq y_i, 1 \leq i \leq n \} |$ where $x, y \in \{0, 1\}$.

In other words, Hamming distance between two codewords is the number of different bits in codewords. An n–cube consists of 2^n nodes where two nodes with binary codewords x and y are connected if the Hamming distance H_{xy} between their codewords is 1.

A *path* in a Q_n is a sequence of connected nodes and the length of the shortest path between nodes *i* and *j* is called the *graphical distance* and is represented as L_{ij}. Let $k = Max[L_{ij} \mid \forall\ i,j, 0 \leq i,j, \leq N-1]$ be the *diameter* of the network. For Q_n, $L_{xy} = H_{xy}$ and $k=n$.

Let V_i^x be the number of processors which are at a graphical distance *i* from a processor *x*. This number is a constant $\forall x \in U$ and is called the *i*–th valency. Then for every processor *x* in Q_n, the valency sequence is given as, $v_i^x = \binom{n}{i}$ for $i = 0, 1, 2 \ldots n$.

Given a set of nodes, *C*, its *graphical covering radius r* in the graph Q_n is defined as: $r = Max_{i \in U}(Min_{j \in C}(L_{ij}))$. In a hypercube topology, every processor has an exactly one diametric processor, that is, the processor which is at distance *k*, the diameter. Such a pair is called *antipodal pair.*

2.1 Partitioning Criterion

Let *C* be the desired set of FCPs and let the sphere assigned to a processor $x \in C$ be denoted by $S_i(x)$, where *i* is the radius of this sphere. The number of processors in $S_i(j)$ is the total number of processors lying at graphical distances 0 through *i*, from processor *x*. The number of processors at the graphical distance *i* is given by valency v_i^x, and the total size of the sphere is $|S_i(x)| = \sum_{j=0}^{i} v_j^x$. The covering radius determines the *range of load re–distribution* used by the FCPs. This range quantifies the graphical distance within which a FCP assigns tasks of a failed processor to the processors of its own sphere, where it is located at the center. The details of the backup copies and scheduling algorithm are described in Section 4. In order to characterize spheres and to describe network partitioning, we need the following definitions.

Intuitively, in a centralized scheme where a single processor acts as an FCP, *i* must be equal to the diameter *(k)* of the network. For our scheme, we are looking for a δ –uniform set *C*, of FCPs, which is the maximal set of processors in Λ, such that the graphical distance among the FCPs is at least δ and $|S_i(x)|$ is constant $\forall x \in C$, where *i* is the covering radius of *C*. For symmetric partitioning, a δ –uniform set *C* of FCPs (for some δ to be determined) is required with identical and graphically symmetric spheres. The size, |*C*|, depends on the selection of δ. If δ is large, the size of |*C*|, is small but spheres are large. If |*C*| is reduced, the sphere size increases and vice versa. In addition, a number of other considerations for the provision of fault–tolerance are given below:

(1) Since, in case of a failure, an FCP needs to re–distribute tasks of the failed processor to all the processors in the sphere, which requires the FCP to maintain state information of load of all the processors within the sphere (the next section regarding load re–distribution), the diameter of the sphere should be as small as possible.

$$
\begin{array}{cc}
\begin{array}{cccccccc}
0&0&0&0&0&0&0&0\\
0&0&0&1&0&1&1&1\\
0&0&1&0&1&1&1&0\\
0&1&0&1&1&1&0&0\\
0&0&1&1&1&0&0&1\\
0&1&1&1&0&0&1&0\\
0&1&1&0&0&1&0&1\\
0&1&0&0&1&0&1&1
\end{array}
&
\begin{array}{cccccccc}
1&1&1&1&1&1&1&1\\
1&1&1&0&1&0&0&0\\
1&1&0&1&0&0&0&1\\
1&0&1&0&0&0&1&1\\
1&1&0&0&0&1&1&0\\
1&0&0&0&1&1&0&1\\
1&0&0&1&1&0&1&0\\
1&0&1&1&0&1&0&0
\end{array}
\\
M & M^C
\end{array}
$$

Figure 1: Hadamard Matrix in 0–1 notation and its complement matrix serving as the binary addresses of primary and secondary FCPs, respectively for Q_8.

$$
\begin{array}{cc}
\begin{array}{ccccccccc}
0&0&0&0&0&0&0&0&0\\
0&0&0&0&1&0&1&1&1\\
0&0&0&1&0&1&1&1&0\\
0&0&1&0&1&1&1&0&0\\
0&0&0&1&1&1&0&0&1\\
0&0&1&1&1&0&0&1&0\\
0&0&1&1&0&0&1&0&1\\
0&0&1&0&0&1&0&1&1
\end{array}
&
\begin{array}{ccccccccc}
1&1&1&1&1&1&1&1&1\\
1&1&1&1&0&1&0&0&0\\
1&1&1&0&1&0&0&0&1\\
1&1&0&1&0&0&0&1&1\\
1&1&1&0&0&0&1&1&0\\
1&1&0&0&0&1&1&0&1\\
1&1&0&0&1&1&0&1&0\\
1&1&0&1&1&0&1&0&0
\end{array}
\\
M & M^C
\end{array}
$$

Figure 2: An extended Hadamard Matrix and its complement matrix serving as the binary addresses of primary and secondary FCPs, respectively for Q_9.

(2) Since an FCP (say, *x*) needs to send/receive $|S_i(x)|$ messages for scheduling of failed–processor tasks within the sphere, the size of the sphere needs to be small because.

For an arbitrary graph, for a given value of $\delta > 2$, finding a uniform set *C* is an NP–hard problem. Determining the minimum sphere size is also NP–hard [8], [18]. Our solution to select the set *C* in these networks uses a combinatorial structure called *Hadamard Matrix which* consists of $\pm$ 1 entries [1]. A *j* by *j* square Hadamard matrix *M* when multiplied with its transpose yields a *jI* matrix, where *I* is the identity matrix. The *complementary Hadamard matrix,* denoted as M^C, is obtained by multiplying all the entries of *M* by –1. Replacing 1 by 0, and –1 by 1, results in a matrix is with 0–1 *notation,* and as denoted as *M*. In Figure 1 a 8×8 Hadamard matrix and its complement are shown. Hadamard matrix of order *n* exists if *n* is 1, 2 or a multiple of 4.

The set C of FCPs for hypercube is selected from the code generated by taking combinations of the rows of Hadamard matrix M and its complement M^c. The set C is also called *Hadamard code*. Specifically, for Q_8, (n being multiple of 4), we take the matrices M and M^c of Figure 1. When $n = 8$, $|C| = 2n$. The set C for other values of n can be obtained using some rules [1]. The summary of rules is as follows. If n mode 4 = 1, we start with the set C obtained from Hadamard matrices M and M^c of size n (Figure 1). The modified set C for the network under consideration can be generated by appending an all 0's and an all 1's column, to M and M^c respectively. If n mod 4 = 2, then the set C is obtained in the same way as the previous case, except we append two columns 0 and 1 to M and 1 and 0 to M^c. However, the all 0's row in M is augmented with bits 00 instead of bits 01. Similarly, the all 1's row in M^c is augmented with bits 11. For the case of n mod 4 = 3, the set C consists of the rows of the truncated matrices M and M^c in 0–1 notation. The truncated matrices (in 0–1 notation) are generated by discarding the all 0's row and column.

2.2 FCP Assignment Rule

Once the set of FCPs in Q_n is known, the rule for the assignment of processors to FCPs requires identifying two types of FCPs, the primary and the secondary FCPs, in order to provide two–level fault–tolerance. Such an assignment is as follows:

The FCPs having the left most bit 0 serve as primary FCPs for those processors which have also left most bit as 0. These FCPs serve as secondary FCPs for the rest of the processors in the network. Similarly, the FCPs having the left most bit as 1 serve as primary FCPs for those processors which have also left most bit as 1 and serve as secondary FCPs for the rest of the processors.

We can notice that this assignment of FCPs to processors is symmetric in the sense that each processor interacts with the same number of primary and secondary FCPs. Also, each FCP manages the same number of processors in terms of providing backup for fault–tolerance and re–distributing the load, if required.

2.3 Processors for Load Re–Distribution

As mentioned earlier, FCPs maintain backups and load status for re–distributing the load to some processors within certain regions in the network, known as spheres. The sphere of an FCP consists of all the processors which are within a distance r from an FCP, where r is the covering radius of C. The number of processors in the sphere, $S_r(x)$, is the total number of processors lying at graphical distances 0 through r, from processor x. Since the number of processors at the graphical distance i is given by valency v_i^x, the total size of the sphere is given as $|S_r(x)| = \sum_{i=0}^{r} v_j^x$.

As mentioned above, the determination of this covering radius is a non–trivial problem. It can be noticed that the processors in one sphere can also be shared by other spheres, depending upon the covering radius r and the graphical distance among FCPs. However, the set C provides the maximal $k/2$ (rad*ius)–uniform set* for Q_n [1]. There are various other advantages of using Hadamard code [1]. Moreover, it is easy to generate a truncated Hadamard matrix (the one without all 1's column) using Symmetric Balanced Incomplete Block Design (SBIBD) [5]. The generator codes, for different values of n–1, can be found using the *difference set approach* [5]. The rest of the blocks (which corresponds to all the elements of the set C, besides codewords with all 0's and all 1's) can be generated by taking n–1 cyclic shifts of such a generator codeword. Table 1 illustrates the generator codewords for various values of n–*1*, which can be used for various Q_n's. For Q_8, the set C can be produced by taking 6 cyclic shifts of code 0010111 (shown in Table 1) and then by appending all 0's row and column to that block [1]. The resultant block is the same as shown earlier in Figure 1 where each row of the matrix represents the binary address of the 16 FCPs.

The set consisting of codewords as given in Figure 1, can also be used to generate the set C for the Q_9 network by appending an all 0's and all 1's column (say at extreme left position), of matrix M and M^c, respectively, as described for case (a). This set is shown in Figure 2. Also, the same set can be used to generate the set C for Q_{10}, as described in the procedure of case (b). The set C for other Q_n 's can be generated by the methods described above.

The topological characteristics and partitioned structures for Q_7, Q_8, Q_9 and Q_{10} networks are summarized in Table II giving the number of processors N, the degree n of each processor, the minimum distance δ between FCPs, the cardinality of the set C, the covering radius r, valencies v_i^x and the size of sphere $|S_t(x)|$, for each network.

3 Failure and Repair Model

As mentioned earlier, failures of components in real life systems are generally random. Also, the failed components, once diagnosed and repaired off–line, can be integrated back into the system. Typically, a processor remains alive most of the time and when it fails, it can be repaired quickly and can become operational. For our study, we assume only the failure of processors in the system. Accordingly, we consider the well known failure/repair model of multiprocessor systems where the availability times as well as the repair times of the processor are assumed to be independent exponential random variables with rates γ and μ_R, respectively. Generally, the ratio of γ/μ_R is assumed to be very small. Since the number of processors (N) in the system is fixed and it has finite population, the cumulative failure and repair rates become state dependent [4]. Accordingly, the Markov model showing the failure and repair processes is depicted in Figure 3. The state of the model represents the number of processors (P) currently operational. At a give time, every operational processor is equally probable to fail. Obviously, there can be more than one processor in the failure mode. The state dependent rate of the failure and repair processes are also shown in Figure 2.

4 Task Re–Allocation Mechanism

In this section, we present the details of the proposed scheme. For this purpose, we describe the system model, the load balancing and load re–distribution algorithms, and then the associated information collection and backup mechanisms.

4.1 Assumptions and Characteristics of the System

The system consists of N processors where each processor is subjected to arrival of tasks. Task scheduling and load balancing are assumed to be completely decentralized, with a distributed scheduling algorithm running on every processor. The tasks arrive at a processor with rate λ tasks/time–unit, which is identical for all the processors. The execution time of a task is assumed to be known. In addition, associated with each task is a deadline. When a task arrives at a processor, the scheduler of that processor tries to guarantee that the task meet it deadline. If this deadline can be met locally, the task is scheduled into the local execution queue which is served on the FCFS principle. If a task cannot meet its deadline locally, it is transferred to another processor. The selection of a remote processor can be done in various ways [13], [14]. However, in this study, we assume that the local scheduler interacts with its immediate neighbors only and gathers their load status. The load status in this case consists of the cumulative unfinished work load which in turn is the sum of the execution times of the tasks waiting in the processor's execution queue plus the remaining execution time of the task which is currently being executed. If a task is to be transferred to another processor, such transfer takes certain amount of time. A task needs to wait in the communication queue of the link until a prior task completes its migration on that link. Since the communication time also counts towards the task's waiting time, the local processor, while making the scheduling decision, also takes into account the communication penalty. The deadline of a task, therefore, consists of its execution time plus the average communication plus some marginal value (D) which depends on the application and is assumed to be supplied by the user.

4.2 Task Replication and Backups at FCPs

In order to provide two level fault–tolerance, the replicated copies of every task are kept as backups on primary and secondary FCPs. The backup queue of a processor is further distributed in a round–robin fashion, first to all the primary FCPs and then again to all the secondary FCPs. Specifically, whenever a task is scheduled at a processor, its copy is sent to the one of the primary FCPs as well as to the associated secondary FCP. The copy of the next task is sent to the next primary FCP present in the round–robin list as well as to the associated secondary FCP. An FCP, therefore, needs to keep backup queues for those $N/2$ processors for which it acts as primary FCP. It also maintains backup queues for the rest of the $N/2$ processors for which it serves as their secondary FCP. It is worth mentioning that in order to provide two level redundancy, there have to be $2N$ redundant copies of N processor queues in the whole system. It is also possible that, at any given time, an FCP(s) may be faulty. In that case, the copy of the task is sent to next available FCP in the round–robin list. In summary, the following task duplication algorithm is executed at each processor:

```
Next FCP = ( Next FCP + 1 ) mod ( Num of FCPs )
While ( Next FCP is faulty )
Do
Next FCP = ( Next FCP + 1 ) mod ( Num of FCPs )
End Do
Send copy of the task to Next FCP
If ( the Secondary FCP is not faulty )
Send the copy of the task to the Secondary FCP
```

Figure 4 shows two processors 00000000 and 11111111 with sixteen FCPs in Q_8. The dotted circles around the FCPs indicate their respective spheres. The identification of these spheres can be found by using the discussion in section 2.3. In this case, the top eight FCPs in the figure corresponding to the binary codewords of matrix M represent the primary FCPs for the processor 00000000, and the eight lower binary codes corresponding to M^c represent secondary FCPs for that processor. On the other hand,

4.3 Load Re–distribution under Failures

When a processor fails, each primary FCP re–schedules the tasks (if it has any any) in its backup queue for the failed processor. Due to real–time constraints, each FCP tries to make sure that the backup tasks still meet their deadlines. An FCP accomplishes this by scheduling each of the backup tasks to the most lightly loaded processors within its sphere. The availability of processors within its sphere provides an FCP with a broader view of the state of the system which enables it to re–distribute the backup load by making better choices in selecting processors. As a result, the backup tasks may still meet their deadlines despite the failure of their original processors. Since the backup queue of the failed processor is itself distributed among multiple FCPs, each FCP needs to schedule a portion of the tasks. The potential benefits offered by this load re–distribution strategy are summarized below:

- The backup queues of processors are themselves load balanced among the FCPs.
- In the case of failures, FCPs are able to select best candidate processors within their respective spheres.
- All the backup tasks can be concurrently re–scheduled in multiple spheres.
- The likelihood of re–allocating tasks to the best possible processors across the whole system is very high since the FCPs are maximally spread in the network.
- Due to the round–robin distribution of tasks in backup queues, the danger of instability due to bulk arrivals at a particular processor or in a particular sphere is greatly reduced.
- From a task's perspective, the combination of its original processor and both FCPs is a unique trio. As a result, the likelihood that this unique trio will fail is low.

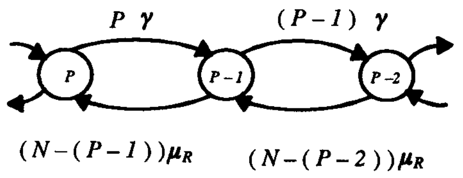

Figure 3: Markov model for failure and repair processes.

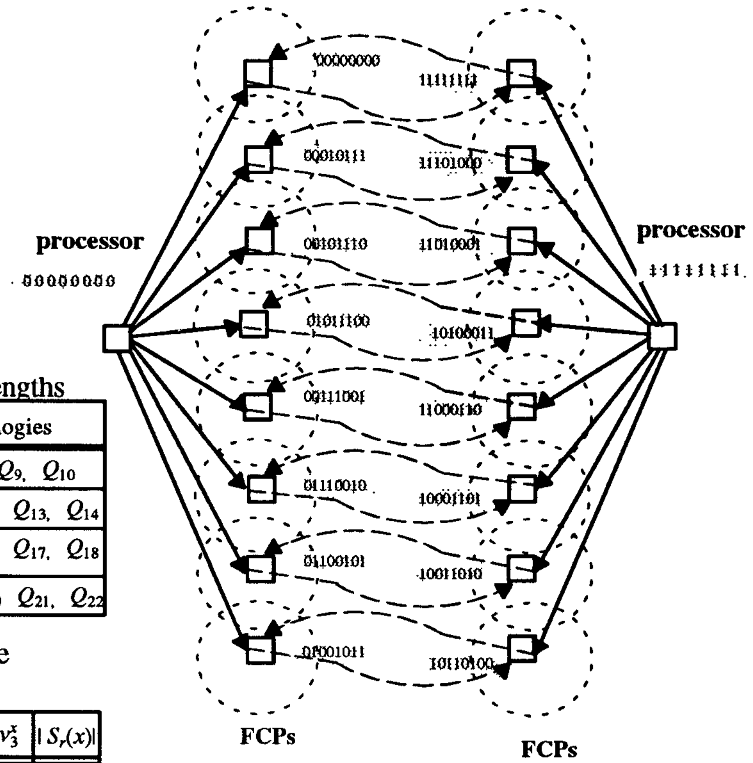

Figure 4: Task replication at FCPs assigned to processor 00000000 and 11111111.

Table 1: Generator codes for different lengths

Length $= n-1$	Generator Codewords	Topologies
7	0010111	Q_7, Q_8, Q_9, Q_{10}
11	10111000101	Q_{11}, Q_{12}, Q_{13}, Q_{14}
15	111101011001000	Q_{15}, Q_{16}, Q_{17}, Q_{18}
19	1001111010100001101	Q_{19}, Q_{820} Q_{21}, Q_{22}

Table II: The characteristics of hypercube networks of various sizes.

Network	N	n	δ	FCPs	r	v_1^r	v_2^r	v_3^r	$\lvert S_r(x)\rvert$
Q_7	128	7	4	14	1	8	–	–	9
Q_8	256	8	4	16	2	8	28	–	37
Q_9	512	9	5	16	2	9	36	–	46
Q_{10}	1024	10	5	16	3	10	45	120	176

Like any fault–tolerant strategy, the proposed strategy also needs to pay some penalties. However, simulation results shown in the later sections indicate that the performance degradation is scalable to the fault occurrence rate and the proposed strategy is adaptive to a wide range of system parameters. The migration of a task from an FCP to a processor within its sphere incurs some communication delay. In addition, we assume that each task is started from the beginning. Nevertheless, this assumption is justified since the issue of task recovery and roll–back incur extra overhead. In addition these issues are not within the scope of this paper and they need to be dealt separately. After re–distributing the load of the faulty processor, all backup queue are deleted. At the same time, the primary FCP informs the corresponding secondary FCP to delete its backup queues for the failed processor. The processor receiving the re–scheduled task treats the backup tasks as a newly arrived task. The processor also sends the backup copy of the re–scheduled task to the FCP. However, the re–scheduled tasks are not allowed to make any further migrations. In spite of two level redundancy, a task can still be lost if:

- Both primary and secondary FCPs of the failed processor have also failed.
- The failed processor is itself an FCP and its secondary FCP has also failed.
- The failed processor is also an FCP and no task backup at the secondary level could be made because the secondary FCP was also faulty at the time the task was scheduled.

However, these events are of very low probability. Even when one of these events does happen, not all but only a fraction of the backup queue of a processor will be lost. It can be noticed that this percentage is of the order of $\Theta(1/n)$. The probability of lost tasks as a result of these events is presented in the next section that proves these claims.

5 Evaluation of the Proposed Fault–Tolerant Strategy

In this section, simulation results for the proposed strategy are presented, for the Q_8 and Q_9 networks. The task arrival process for this study been modeled as a Poisson process with average arrival rate of λ tasks/unit–time which is identical for all processors. The execution and migration times of tasks have been assumed to be exponentially distributed with a mean of $1/\mu_E$ time–units/task and $1/\mu_c$ time–units/task, respectively. The task deadline has been computed by generating a random number from a uniform distribution with an average of D time–units. The processor failure and repair rates are also assumed to be exponentially distributed with a mean of $1/\gamma$ time–units/processor and $1/\mu_R$ time–units/processor, respectively. Failure and repair rates are independent with respect to the rest of the system parameters. All results are presented with 95 percent confidence interval, with the size of the interval varying up to plus or minus 5 percent of the sample mean. The first performance measure is the missing probability, defined as the probability that a task does not finish its execution within its specified deadline [14]. In our study, the impacts of average task deadline (D) and three important system parameters, including the frequency of processor failures (γ), system load (λ/μ_E) and task migration rate (μ_C), on deadline missing probability are evaluated. Since γ and $\mu_R$alone doe not present a clear view of the number of faulty processors at any given time because of the continuous failure and repair processes, we do not present our results in terms of these two parameters. Rather, the results presented in the next section are described with respect to the average number of faulty processors. In other words, for given values of γ and μ_R, the steady–state value of average number of faulty processors in the system becomes fixed which is given by $N\gamma/(\gamma+\mu_R)$. In simulation, we have used different values of γ and have kept μ_R fixed as 0.05.

5.1 Impact of Deadline and Frequency of Failures

The performance of the underlying decentralized scheduling clearly depends on the specified deadlines of tasks. Recall that the deadline of a task in simulation is computed by adding a task's execution time to its associated value of D and $1/\mu_c$. Since D is a uniformly distributed random variable, the minimum value of D is 0 while the maximum value is 2 times the average value. With tight deadlines (a low value of D), more tasks are likely to miss their deadlines where-

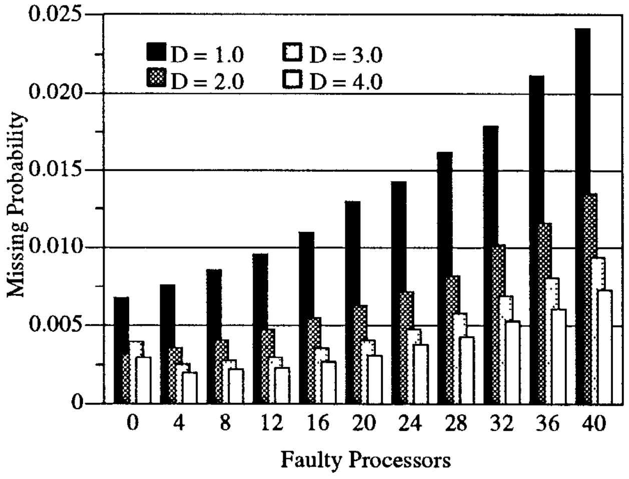

Figure 5: Deadline missing probability versus average number of faulty processors at various values of D for the Q_8 network.

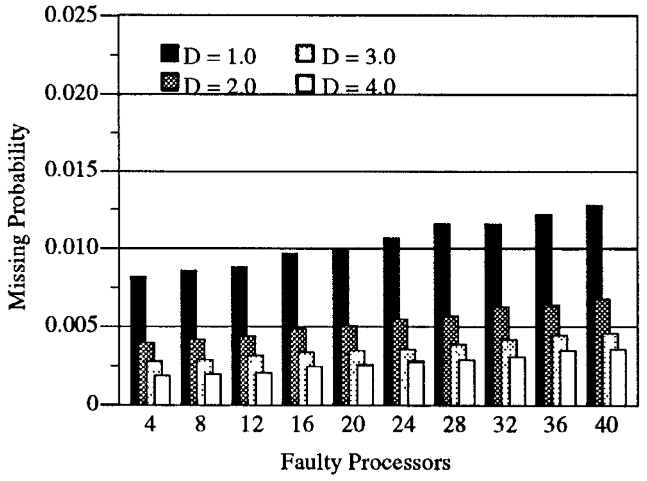

Figure 6: Deadline missing probability versus average number of faulty processors at various values of D for the Q_9 network.

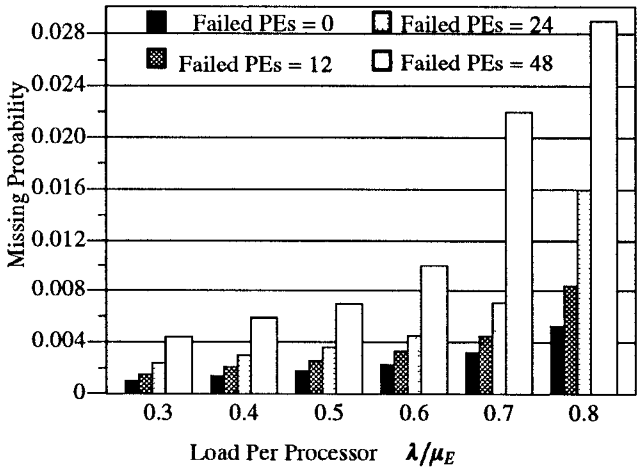

Figure 7: The effects of system load and fault rate on the deadline missing probability for the Q_8network.

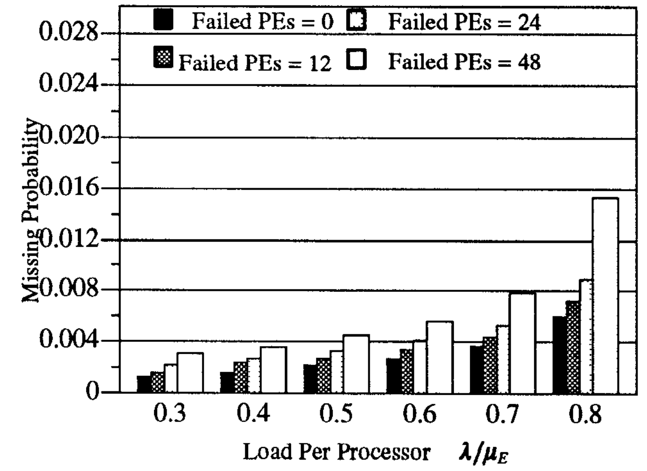

Figure 8: The effects of system load and fault rate on the deadline missing probability for the Q_9 network.

as loose deadlines (a higher value of D) imply that more tasks will meet their deadlines, even when the system is fault–free. The occurrence of failures further degrades the deadline missing probability. The frequency of failures affects the amount of load which is re–injected back into the system. Recall that in the proposed strategy, the load submitted to the faulty processors, while they are under repair, is not rejected; rather, new tasks at the faulty processor are assigned to primary FCPs. Therefore, in addition to load re–distribution, FCPs are also subjected to some additional load. The results presented in this section show the effects of task deadlines and processor failure frequencies.

The results indicating the deadline missing probability are shown in Figure 5 and Figure 6, for Q_8 and Q_9, respectively. Four average values of D, 1.0, 2.0, 3.0 and 4.0 have been selected while the number of faulty processor has been varied from 4 to 40. The non–failure case (faulty processors equal to 0) is also included for comparison. The task arrival rate per processor (γ) is 0.7, and communication rate (μ_C) is set to be 20 tasks/time–unit. We notice that if the average number of faulty processors increases, the missing probability also increases because not only the tasks (if any) waiting in the execution queues of the failed processors have to be re–scheduled but also the tasks being executed at the time of failure have to be aborted and re–scheduled. These results indicate that the proposed strategy can sustain negligible degradation in performance for different combinations of D and γ. For example, in the case of Q_8 with D equal to 1 as shown in Figure 5, the difference in the missing probability at no–failure case and 18 faulty processors case is 0.017. These results are encouraging, given that 40 of the 256 processors are on the average out of order while their backup tasks as well as new tasks are also being accommodated. When D is equal to 1, this difference 0.036. This indicates that the proposed strategy is able to tolerate failures under both strict ($D = 1$) and relaxed ($D = 4$) deadlines.

The result for Q_9 are more encouraging as can be noticed from Figure 6 which indicates that although the missing probabilities for the no–failure case are almost the same as those of Q_8, these probabilities are considerably less for the failure conditions. This is due to the large size of Q_9 which has a sphere size of 46 as compared to 37 for Q_8. The larger sphere size of Q_9 enables an FCP to re–schedule a backup task at a more suitable processor.

5.2 Impact of System Parameters

Two other important factor that can have a significant impact on normal load distribution as well as load re–distribution are the system load and task migration time. A fault–tolerant strategy should be able to perform equally well if these factors happen to change. The results presented in this section examine the effects of these factor under failure and non–failure conditions. The missing probability is obtained by varying λ on each processor from 0.3 to 0.8, for both networks. Recall that throughout this study μ_E is kept as 1 and therefore the system load ($\varrho = \lambda/\mu_E$) corresponds to λ. D and μ_C have been selected to be 2 and 20, respectively. For failures, three failure conditions corresponding to 12, 24 and 48 faulty processors have been considered Q_8 while these numbers are doubled for the Q_9 network. The deadline missing probabilities are shown in Figure 7 and Figure 8 for Q_8 and Q_9, respectively.

The deadline missing probability, obviously, depends on the system load, under both failure or non–failure conditions. The important point to notice from these curves is that under any load conditions the performance degradation due to failures does not strictly dependent on the system load. This can be observed from Figure 7 and Figure 8 where the deadline missing probabilities are shown to increase slightly if load is varied from 0.3 to 0.8 for both Q_8 and Q_9. For Q_8, the deadline missing probability increases with a noticeable difference only when load is equal to 0.8. For Q_9, this difference is even smaller.

The results presented next have been obtained by considering different task migration costs. For these result, μ_C has been varied from 4 to 40 tasks/unit–time implying a slower to faster communication network. The values of D and λ have been kept as 2.0 and 0.8, respectively, and the average number of faulty processors has been selected to be the same as shown in earlier in Figure 9 and 10. The deadline missing probabilities for Q_8 and Q_9 are illustrated in Figure 9 and Figure 10, respectively. In the no–failure case, the task migration time affects the processor to processor task migration for normal load balancing. On the other hand, in the failure case, the migration overhead is incurred when an FCP re–schedules a task within its sphere. Therefore, in both cases, the successful completion of a task before its deadline is affected if this overhead is high. From Figure 9 and Figure 10, it can be noticed that, under normal operation, the performance of both networks is almost identical. The increase in deadline missing probability is negligible, except when the faulty processors are equal to 40. Nevertheless, the proposed strategy maintains this slight degradation in performance over the complete range of μ_C.

5.3 Robustness of Fault–Tolerance

So far the results presented have shown that the proposed strategy is able to adaptive to system parameters such as D, λ and μ_C. The degradation in the overall performance of deadline missing probability, as compared to that of fault–free environment, is due to achieving our main objective of saving those tasks which would have been lost if there were no fault–tolerance and load re–distribution. The next question is how does a re–scheduled task perform in terms of meeting its deadline. Note that, we do not consider task level roll back and recovery, and we assume that a backup task starts it execution from the beginning. Therefore a backup task starts all over again even if it was under processing at the time its original processors crashed. However, the task retains its original deadline and is required to finish its execution before that time. The obvious question is how good is the strategy in achieving these objectives. The usefulness of such a fault–tolerant strategy, therefore, should be judged by observing its ability to minimize the loss of tasks and its efficiency in quickly re–scheduling backup tasks. For this purpose, we have observed from simulation the deadline meeting probability of a re–scheduled task and the probability of a task being totally lost.

The success probability of re–scheduled tasks should be distinguished from the overall probability of all the tasks (as shown in previous sections) since this probability has been obtained by observing only the re–scheduled tasks from the backup queue. Figure 11 and Figure 12 show these probabilities for the two networks under different levels of failures. The load per processor is set to be 0.8 and task migration rate is equal to 20. We observe from these figures that the success probability of re–scheduled tasks is affected by D and not by the level of failure. For the Q_8 network, this probability is about 0.4 when D is 1.0. The probability becomes greater than 0.6 when D is 4, and is even higher for Q_9. This reasonably high success probability is due to the fact that it is very likely that there are idle or lightly loaded processors in the system. The load re–distribution mechanism enables an FCP to re–allocate backup on the idle processors within its sphere. Since the FCPs are spread across the whole system and the backup queues of the failing processors are spread across those FCPs, the backup tasks are re–scheduled to idle or lightly loaded processors even when those processors are graphically located at different places in the network topology.

As can be noticed from the proposed partitioning scheme (section 2), the number of FCPs are always of O(*log*

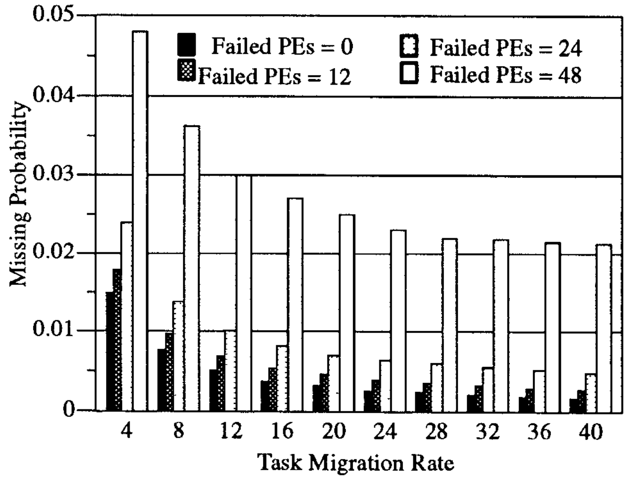

Figure 9: The effects of task migration and fault rate on the deadline missing probability for the Q_8 network.

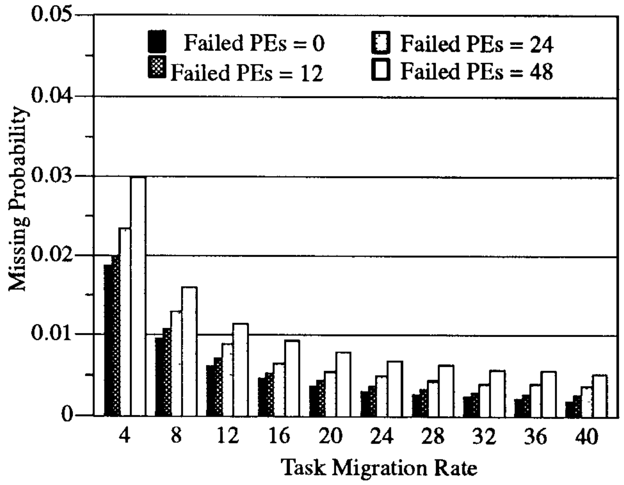

Figure 10: The effects of task migration and fault rate on the deadline missing probability for the Q_9 network.

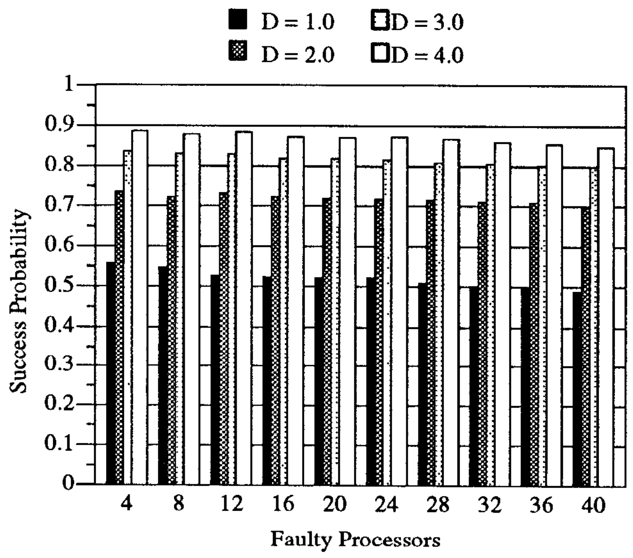

Figure 11: The success probability of re–scheduled tasks for the Q_8network

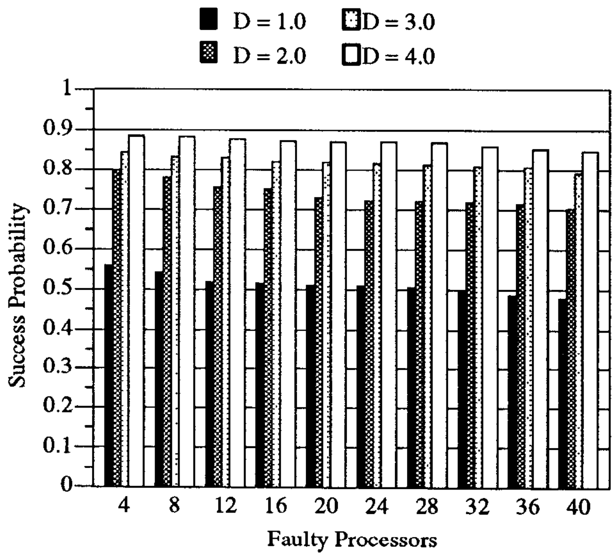

Figure 12: The success probability of re–scheduled tasks for the Q_9 network

Table III Percentage of task which still meet their deadlines after re-scheduling

Faulty PEs	Q_8	Q_9
4	2.04×10^{-6}	1.00×10^{-6}
8	5.96×10^{-5}	1.56×10^{-6}
16	3.28×10^{-4}	1.12×10^{-5}
24	7.22×10^{-4}	5.02×10^{-5}
32	5.40×10^{-4}	2.35×10^{-5}
40	1.81×10^{-3}	1.82×10^{-4}
48	3.88×10^{-3}	2.73×10^{-3}

N). Therefore, as the network size increases, the sphere size per FCP increases. Due to this reason, Q_9 is shown to perform consistently better than Q_8. Accordingly, the proposed scheme is expected to perform better for larger systems.

Table III provides the probability of a task being lost which could not be executed at all due to the failures of the processor as well as the primary and the secondary FCPs. In other words, the entries in the table provide the probability of task being lost altogether. Generally, this probability depends on two factors. First, the level of redundancy which is two in this case and the number of FCPs among which the tasks are distributed in a round-robin fashion. As can be noticed, the values shown in Table III are considerably low.

6 Conclusions

In this paper, we have proposed a new fault-tolerant approach for large-scale hypercube multicomputer systems. The use of Hadamard matrix results in an efficient strategy for identifying fault control processors and for partitioning these systems for load re-distribution. The central processors of the spheres, called fault-control processors, provide a two-level task redundancy and efficiently re-distribute the load of failed processors within their spheres. For failure and repair processes, we have assumed a realistic failure-repair system environment. In addition, we have taken into account the load submitted to processors while they are under repair. The performance of the proposed strategy has been evaluated for both failure and no-failure cases. The degradation in the overall performance is due to achieving the objective of saving those tasks which would have lost if there were no fault-tolerance and load re-distribution support. It is shown that, using the proposed strategy, the probability of meeting deadlines by re-scheduled tasks is reasonably high. The probability of a task being lost due to multiple failures has also been shown to be negligible.

References

[1] I. Ahmad and A. Ghafoor, "A Semi-Distributed Distributed task Allocation Strategy for large Hypercube Supercomputers " Proc. of *Supercomputing '90*, New York, Nov. 1990, pp 898–907.

[2] Y. Chang and K. G. Shin, "Load Sharing in Hypercube Multicomputers in the Presence of Node Failures," in Proc. of *Fifth Distributed Memory Computing Conference, Vol. II*, April 1990, pp. pp. 1465–1474.

[3] T. C. K. Chou and J. A. Abraham, "Load ReDistribution Under Failure in Distributed Systems", *IEEE Trans. on Computers*, vol. C-32, no. 9, Sept. 1983, pp. 799–808.

[4] C. R. Das, J. T. Kreulen and M. J. Thazhuthaveetil, "Dependability Modeling for Multiprocessors", *IEEE Computer*, Oct. 1990, pp. 7–19.

[5] M. Hall Jr., *Combinatorial Theory*, 2nd Ed., John Wiley and Sons, New York, 1986.1

[6] J. F. Kurose and R. Chipalkatti, "Load Sharing in Soft Real-Time Distributed Computer Systems," *IEEE Trans. on Computers*, vol. C-36, no. 8, August 1987, pp. 993–1000.

[7] D. W. Leinbaugh and M. Yamini, "Guaranteed Response Times in a Distributed Hard-Real-Time Environment," *IEEE Trans. on Software Eng.*, vol. SE-12, Dec. 1986, pp. 1139–1144.

[8] A. M. McLoughlin, "The complexity of Computing the Covering Radius of a Code," *IEEE Trans. on Information Theory*, col IT-30, Nov., 1984, pp. 800–804.

[9] F. J. MacWilliams and N. J. A. Sloane, *The Theory of Error-Correcting Codes, vols. I and II*, New York: North Holland, 1977.

[10] J. K. Muppala, S. P. Woolet and K. S. Trivedi, "Real-Time-Systems Performance in the Presence of Failures, " *IEEE Computer*, May 1991, pp. 37–47.

[11] B. A.A. Nazief, "Empirical Study of Load Distribution Strategies on Multicomputers, " Ph.D Dissertation, University of Illinois at Urbana-Champaign, September 1991.

[12] X. Qian and Qing Yang, "Load Balancing on Generalized Hypercube and Mesh Multiprocessors with LAL" *Proc. of The 11-th Int'l conf. on Distributed Computing systems*, May 1991, pp. 402–409.

[13] K. Ramamritham, J. A. Stankovic and W. Zhao, "Distributed Scheduling of Tasks with Deadlines and Resource Requirements," *IEEE Trans. on Computers*, vol. 38, no. 8, Aug. 1989, pp. 1110–1123.

[14] K. G. Shin and Y.-C. Chang, "Load Sharing in Distributed Real-Time Systems with State-Change Broadcasts," *IEEE Trans. on Computers*, vol. 38, no. 8, Aug. 1989, pp. 1124–1142.

[15] N. G. Shivrati and M. Singhal, "A Transfer Policy for Global Scheduling Algorithms to Schedule tasks with Deadlines, " in Proc. of *11-th Int'l. Conf. on Distributed Computing Systems*, May 1991, pp. 248–255.

[16] P. J. Skerrett, "Future Computers: The Teraflops Race, " *Popular Science*, March 1992.

[17] J. A. Stankovic, "Decentralized Decision Making for Task Allocation in a Hard Real-Time System," *IEEE Trans. on Computers*, vol. 38, no. 3, March 1989, pp. 341–355.

[18] L. J. Stochmeyer and V. V. Vazirani, "NP-Completeness of some Generalization of the Maximum Matching problems," *Information Proc. Letters*, vol. 15, 1982, pp 14–19.

[19] Thinking Machines, *CM5, Technical Summary*, October 1991.