3 CONCLUSION

In this paper, we analyzed the harvest rate of reconfigurable multipipeline processor arrays. We showed that the "shifting" or "fault stealing" phenomenon during reconfiguration can be described as the maximum weighted chains in a poset with random weights, and we used a combinatorial argument to give a bound on the size of the maximum weighted chain. Our method is the first purely analytical approach to analyzing reconfiguration of linear arrays. We propose as an open problem to find the exact value of $h(m, n)$.

ACKNOWLEDGMENTS

The authors thank Douglas B. West for valuable discussions concerning this research, Ran Libeskind-Hadas for suggestions, Harry Kesten, Peter Winkler for providing references, and an anonymous referee (D) for improving the presentation. This research was supported in part by the SDIO/IST and managed by the U.S. Office of Naval Research under contract N00014-89-K-0070. Weiping Shi was also supported in part by the U.S. National Science Foundation under grant MIP-9309120.

REFERENCES

[1] M. Aigner, *Combinatorial Theory*. Springer-Verlag, 1979.
[2] J.W. Greene and A. Gamal, "Configuration of VLSI Arrays in the Presence of Defects," *J. ACM*, vol. 41, no. 4, pp. 694–717, 1984.
[3] G. Grimmett, *Percolation*. New York: Springer-Verlag, 1989.
[4] G. Grimmett and H. Kesten, "First Passage Percolation, Network Flows and Electrical Resistances," *Z. Wahrscheinlichkeitstheor.* Verw 66, pp. 335–366, 1984.
[5] R. Gupta, A. Zorat, and I. V. Ramakrishnan, "Reconfigurable Multipipelines for Vector Supercomputers," *IEEE Trans. Computers*, vol. 38, no. 9, pp. 1,297–1,307, Sept. 1989.
[6] K. Hwang, *Advanced Computer Architecture: Parallelism, Scalability, Programmability*. New York: McGraw-Hill, 1993.
[7] F.T. Leighton and C.E. Leiserson, "Wafer-Scale Integration of Systolic Arrays," *IEEE Trans. Computers*, vol. 34, no. 5, pp. 448-461, May 1985.
[8] R. Libeskind-Hadas, "Reconfiguration of Fault Tolerant VLSI Systems," PhD thesis, Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign, Oct. 1993.
[9] J.S. Provan and M.O. Ball, "The Complexity of Counting Cuts and of Computing the Reliability That a Graph Is Connected," *SIAM J. Computing*, vol. 12, no. 4, pp. 777–788, Nov. 1983.
[10] W. Shi, "Design, Analysis and Reconfiguration of Defect-Tolerant VLSI and Parallel Processing Arrays," PhD thesis, Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign, June 1992. Coordinated Science Laboratory Technical Report CRHC-94-21, Sept. 1994.
[11] W.S. Stornetta, B.A. Huberman, and T. Hogg, "Scaling Theory for Fault Stealing Algorithms in Large Systolic Arrays," *IEEE Trans. Computer-Aided Design*, vol. 9, no. 3, pp. 290–298, Mar. 1990.

Load Sharing in Hypercube-Connected Multicomputers in the Presence of Node Failures

Yi-Chieh Chang and Kang G. Shin, *Fellow, IEEE*

Abstract—This paper addresses two important issues associated with load sharing (LS) in hypercube-connected multicomputers: 1) ordering fault-free nodes as preferred receivers of "overflow" tasks for each overloaded node and 2) developing an LS mechanism to handle node failures. Nodes are arranged into *preferred lists* of receivers of overflow tasks in such a way that each node will be selected as the *k*th preferred node of one and only one other node [1]. Such lists are proven to allow the overflow tasks to be evenly distributed throughout the entire system. However, the occurrence of node failures will destroy the original structure of a preferred list if the failed nodes are simply dropped from the list, thus forcing some nodes to be selected as the *k*th preferred node of more than one other node. We propose three algorithms to modify the preferred list such that its original features can be retained regardless of the number of faulty nodes in the system. It is shown that the number of adjustments or the communication overhead of these algorithms is minimal. Using the modified preferred lists, we also proposed a simple mechanism to tolerate node failures. Each node is equipped with a backup queue which stores and updates the information on the tasks arriving/completing at its most preferred node.

Index Terms—Load sharing, hypercube-connected multicomputers, real-time systems, node failures, backup queues.

◆

1 INTRODUCTION

LOAD sharing (LS) in general-purpose distributed systems has been studied extensively by numerous researchers and many LS algorithms proposed [2], [3], [4], [5]. These LS algorithms are usually designed to minimize the average task-response time. By contrast, LS in distributed real-time systems has been addressed far less than that in general-purpose distributed systems.

In [6], we have proposed a decentralized, dynamic LS method for real-time applications. In this method, each node maintains the state of a set of nodes in its proximity, called a *buddy set*. Three thresholds of queue length (QL), denoted by TH_u, TH_f, and TH_v, are used to define the (load) state of a node. A node is said to be *underloaded* if $QL \leq TH_u$, *medium-loaded* if $TH_u < QL \leq TH_f$, *fully-loaded* if $TH_f < QL \leq TH_v$, and *overloaded* if $QL > TH_v$. Whenever a node becomes fully-loaded due to the arrival and/or transfer of tasks, it will broadcast this change of state to all the nodes in its buddy set; so will it when a node becomes underloaded as a result of completing the execution of tasks. Every node that receives this state-change broadcast will update its state information by marking the node as fully-loaded or underloaded in its ordered list (called a *preferred list*) of available receivers. When a node becomes overloaded, it can then select, without probing other nodes, the first underloaded node from its preferred list. Note that the preferred list of each node does not change over the time, but the nodes will be dynamically marked as underloaded or overloaded according to their load states, so that an overloaded node may select the first underloaded node from its preferred list.

• *The authors are with the Real-Time Computing Laboratory, Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109-2122. E-mail: kgshin@eecs.umich.edu.*

Manuscript received July 11, 1994; revised July 10, 1995.
For information on obtaining reprints of this article, please send e-mail to: transcom@computer.org, and reference IEEECS Log Number C95165.

0018-9340/96$05.00

Two most important issues in constructing preferred lists and buddy sets are identified as the *coordination* and *congestion* problems [1]. First, when the number of overloaded nodes is not greater than the number of underloaded nodes, no more than one overloaded node should be allowed to select the same underloaded node as a receiver; otherwise, an underloaded node could become overloaded due to the simultaneous transfer of overflow tasks from multiple overloaded nodes, even if there are other underloaded nodes in the system. This problem results from lack of coordination among overloaded nodes. Second, in order to minimize the task transfer delay, the buddy set of a node is composed of those nodes in its physical proximity (e.g., those one or two hops away). The congestion problem arises when a *hot region*—a region where the number of overloaded nodes is greater than that of underloaded nodes—is formed in the system. To resolve this problem, the overloaded nodes in a buddy set should be able to transfer their overflow tasks to the nodes in different buddy sets such that the tasks arriving at overloaded nodes within a hot region can be shared throughout the entire system, not just by those nodes in the same buddy set.

We have already developed an algorithm to generate the preferred lists in hypercube-connected multicomputers *in the absence of* faulty nodes [1]. Preferred lists are so constructed that each node will be selected as the kth preferred node of one and only one other node. In order to reduce the communication overhead for employing the LS method, the buddy set is chosen as the first few nodes in a preferred list. In an early paper [6], we also showed that the buddy set sizes of 10 to 15 nodes perform well for a system with up to 1,024 nodes. Moreover, we showed in [1] that the coordination and congestion problems can be resolved effectively with such preferred lists. However, occurrence of node failures will destroy the original structure of a preferred list if faulty nodes are simply dropped from the preferred list. For example, let N_x and N_y be the most and second preferred nodes of N_1, and let N_y be the most preferred node of N_2. If N_x becomes faulty and is dropped from N_1's preferred list, then N_y will become the most preferred node of *both* N_1 and N_2, thus losing the original property that a node can be selected as the kth preferred node by one and only one other node. The same argument also applies to the case when both N_x and N_2 are overloaded, but such a case, even if it occurs, should not last long; otherwise, the two nodes will be intrinsically unstable. Thus, "static node pairing" should not be altered to deal with natural load fluctuations.

For the reasons discussed above, we need to develop an algorithm to modify the preferred lists in case of node failures so that the original features of LS may be retained. We will show that such a modification/adjustment is always possible regardless of the number of faulty nodes in the system. Moreover, we will propose three adjustment algorithms which will be shown to incur only minimal communication overhead.

If a node becomes faulty before completing all tasks in its queue, all of the unfinished tasks in the queue will be lost unless some fault-tolerant mechanisms are provided. Using the modified preferred lists, a simple fault-tolerant mechanism can be used to avoid/minimize task losses as follows. Each node N_x is equipped with a backup queue for the tasks at its most preferred node N_a which is in turn equipped with a backup queue for N_x. Whenever N_x fails, its most preferred node N_a will process the unfinished tasks in its backup queue. If this node gets overloaded, it can transfer them just like those tasks arriving at the node. By using the proposed algorithms, the failed node in the preferred lists will be replaced by a fault-free node such that the node, which originally selected the failed node as its most preferred node, will always be backed up by a fault-free node.

There are two advantages of using an existing preferred list to back up failed nodes. First, using a preferred list incurs no extra cost in providing the backup queue(s) for each node. Second, the proposed modification algorithms ensure faulty nodes to be removed from the preferred list, so that as long as a node is not isolated in the system, it can always find a fault-free node to back up its own tasks.

The rest of this paper is organized as follows. Some important features of constructing preferred lists are briefly reviewed in Section 2. We propose in Section 3 the adjustment algorithms and fault-tolerant mechanisms whose performance is evaluated via modeling and simulation in Section 4. The paper concludes with Section 5.

2 CONSTRUCTION OF PREFERRED LISTS

The time to transfer a task is usually proportional to the distance between the two nodes involved, so the preferred list of each node is constructed based on inter-node distances. The mth *component group* of N_i's preferred list is composed of those nodes m hops away from N_i where $1 \leq m \leq n$ and n is the dimension of the *binary* hypercube under consideration. Note that N_i's preferred list is an ordered set of all the other nodes on the system. Let N_i's address be represented by $i_{n-1}i_{n-2} \cdots i_0$ and let I_k denote an n-bit number, all but the kth bit of which are zeros. The symbol $\oplus$ denotes the bitwise EXCLUSIVE-OR operation. The nodes of N_i's preferred list are then determined as follows:

DEFINITION 1.

1) The nodes in the first component group are ordered as $\{(i_{n-1}i_{n-2} \cdots i_0) \oplus I_j\}$ $(j = 0, 1, \ldots, n-1)$.
2) The nodes in the second component group are ordered as $\{(i_{n-1}i_{n-2} \cdots i_0) \oplus I_j \oplus I_k\}$ $(j = 1, \ldots, n-2, 0, \text{and } j+1 \leq k \leq n-1)$.
3) The nodes in the third component group are ordered as $\{(i_{n-1}i_{n-2} \cdots i_0) \oplus I_j \oplus I_k \oplus I_\ell\}$ $(j = 1, \ldots, n-3, 0, j+1 \leq k \leq n-1, \text{and } k+1 \leq \ell \leq n-1)$.
4) In general, the nodes in the kth component group of N_i's preferred list are ordered as $\{(i_{n-1}i_{n-2} \cdots i_0) \oplus I_{j1} \oplus I_{j2} \cdots \oplus I_{jk}\}$ $(j_1 = 1, \ldots, n-k, 0, j_1+1 \leq j_2 \leq n-1, \cdots, \text{and } j_{k-1}+1 \leq j_k \leq n-1)$.

An example of preferred lists generated for a 4–cube, or Q_4, is shown in Fig. 1. N_i's buddy set of size σ is formed with the first σ nodes in N_i's preferred list.

Let **N** denote the set of nodes in the system. Then, in order to describe the properties of a preferred list, it is necessary to introduce the following notation and the (forward) node mapping function M_j: $\mathbf{N} \rightarrow \mathbf{N}$ and the inverse node mapping function $M_j^{-1} : \mathbf{N} \rightarrow \mathbf{N}$, such that $M_j(N_i)$ is the jth preferred node of N_i, and $M_j^{-1}(N_i)$ is the node that selects N_i as its jth preferred node. The forward and inverse mapping function can be applied recursively to identify any node in a buddy set. For example, $M_k(M_j(N_i))$ is the kth preferred node of $M_j(N_i)$, but $M_k^{-1}(M_j(N_i))$ is the node that selects the jth preferred node of N_i as its kth preferred node. Using the preferred list as shown in Fig. 1, a few more examples of using the node mapping function are $M_2(M_1(N_0)) = N_3$, $M_3^{-1}(M_1(N_0)) = N_5$, $M_3(M_2^{-1}(N_0)) = N_6$, and $M_3^{-1}(M_2^{-1}(N_1)) = N_7$.

- S_{n_L} : N_i's buddy set of size σ, i.e., $S_{N_i} = \{M_1(N_i), M_2(N_i), \ldots, M_\sigma(N_i)\}$.
- $\overline{S_{N_i}}$: The ordered set that includes all nodes in the N_i's preferred list except those nodes in N_i's buddy set, i.e., $\overline{S_{N_i}} = \{M_{\sigma+1}(N_i), M_{\sigma+2}(N_i), \ldots, M_{2^n-1}(N_i)\}$.

Order of preference	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
N_0	1	2	4	8	6	10	12	3	5	9	14	13	11	7	15
N_1	0	3	5	9	7	11	13	2	4	8	15	12	10	6	14
N_2	3	0	6	10	4	8	14	1	7	11	12	15	9	5	13
N_3	2	1	7	11	5	9	15	0	6	10	13	14	8	4	12
N_4	5	6	0	12	2	14	8	7	1	13	10	9	15	3	11
N_5	4	7	1	13	3	15	9	6	0	12	11	8	14	2	10
N_6	7	4	2	14	0	12	10	5	3	15	8	11	13	1	9
N_7	6	5	3	15	1	13	11	4	2	14	9	10	12	0	8
N_8	9	10	12	0	14	2	4	11	13	1	6	5	3	15	7
N_9	8	11	13	1	15	3	5	10	12	0	7	4	2	14	6
N_{10}	11	8	14	2	12	0	6	9	15	3	4	7	1	13	5
N_{11}	10	9	15	3	13	1	7	8	14	2	5	6	0	12	4
N_{12}	13	14	8	4	10	6	0	15	9	5	2	1	7	11	3
N_{13}	12	15	9	5	11	7	1	14	8	4	3	0	6	10	2
N_{14}	15	12	10	6	8	4	2	13	11	7	0	3	5	9	1
N_{15}	14	13	11	7	9	5	3	12	10	6	1	2	4	8	0

Fig. 1. Preferred lists in a 4-cube system.

TABLE 1
NUMBER OF NODES SELECTED BY MORE THAN TWO NODES AS THEIR *k*TH PREFERRED NODES IN AN 8-CUBE SYSTEM WITH A BUDDY SET OF 10 NODES, WHERE $1 \le k \le 10$

Preference # of faulty nodes	1	2	3	4	5	6	7	8	9	10
1	1	2	3	4	5	6	7	8	7	8
2	2	4	6	8	10	12	14	12	12	14
3	3	6	9	12	15	18	21	16	16	20
12	10	19	25	29	36	40	43	60	45	46
38	30	42	47	49	58	55	60	62	58	59
64	36	46	52	51	51	53	52	51	54	52

- $S_{N_i}^{-1}$: The ordered set of $M_k^{-1}(N_i)$s, $1 \le k \le \sigma$, i.e.,

$$S_{N_i}^{-1} = \{M_1^{-1}(N_i), M_2^{-1}(N_i), \ldots, M_\sigma^{-1}(N_i)\}.$$

The following theorems and corollaries state that the preferred lists designed above can solve both the coordination and congestion problems in a failure-free situation. (See [1] for their detailed account).

THEOREM 1. *Each node in a Q_n will be selected as the kth preferred node of one and only one other node, i.e., for any N_i, $M_j^{-1}(N_i) \ne M_k^{-1}(N_i)$, $\forall j, k \in \{1, \ldots, \sigma\}$ and $j \ne k$.*

Since each node in a hypercube is selected as the most preferred node by one and only one other node, the probability of an underloaded node being selected by more than one overloaded node is very small, thereby solving the coordination problem.

THEOREM 2. *If $\sigma \le \binom{n}{2}$, the most preferred node of each node in a buddy set must come from a different buddy set. In other words, a node and its most preferred node are not co-located in the same S_{N_i} for any N_i.*

Since each node will be selected as the most preferred node by one and only one other node, the probability of an underloaded node being selected by more than one overloaded node is very small, thereby solving the coordination problem. Furthermore, since an overloaded node is most likely to transfer an overflow task to its most preferred node, the overloaded nodes in a buddy set will spread their overflow tasks out to many different buddy sets instead of overloading the nodes in its own buddy set, thus solving the congestion problem.

3 LOAD SHARING IN THE PRESENCE OF NODE FAILURES

Faulty nodes are assumed to not affect the operation of fault-free nodes in the system. Node failures are detected by the other nodes through communication timeouts, and testing issues are outside of the scope of this paper. Adjusting the preferred lists and implementing a fault-tolerant backup queue are discussed below. All the adjustment algorithms will be applied only to those nodes that remain connected in the presence of faulty nodes or links. That is, we will not consider "isolated" nodes.

3.1 Adjusting the Preferred Lists

If faulty nodes are simply dropped from the preferred lists, some nodes will be selected "permanently" by more than one node as the ℓth preferred nodes where $1 \le \ell \le \sigma$. For example, suppose node N_0 is faulty, then the preferred lists of the nodes in $S_{N_0}^{-1}$ will be changed as follows. Since N_0 is faulty, the node that selects N_0 as its *j*th preferred node must select its $(j + 1)$th node to replace N_0. So, $M_{j+1}(M_j^{-1}(N_0))$ will become $M_j(M_j^{-1}(N_0))$ for $j = 1, \ldots, \sigma$. However, according to Theorem 1, the $(j + 1)$th preferred node of $M_j^{-1}(N_0)$ is already assigned as the *j*th preferred node of another node, so this node will be selected by two different nodes as the *j*th preferred node if N_0 is simply dropped from the preferred list of $M_j^{-1}(N_0)$.

Generally, if there are *f* faulty nodes none of which are located in the same buddy set, then there will be $f \times k$ nodes to be selected as the *k*th preferred node by two other nodes. However, if some faulty nodes are located in the same buddy set, the number of nodes that will be selected by more than two nodes is very complicated (thus difficult) to derive. For example, Table 1 lists the number of nodes that are selected by more than two nodes in a Q_8.

If a node were selected by more than one node as their most preferred node, the failure of this node will affect all of them. Thus, it is desirable to adjust the preferred lists dynamically whenever a node becomes faulty, such that Theorems 1 and 2 will remain valid even in case of node failures. The following theorem states that such an adjustment is always possible, but not unique.

THEOREM 3. *Regardless of the number of faulty nodes in the system, there always exists an algorithm to adjust the preferred lists, such that Theorems* 1 *and* 2 *will hold.*

PROOF. Let S_g be the ordered set of fault-free nodes in the system. Suppose $|S_g| = x$, and let $N^{(i)}$ be the ith node in S_g. $N^{(1)}$ can choose any node other than itself (in $x - 1$ ways) as its most preferred node, $N^{(2)}$ can choose any node other than itself (in $x - 2$ ways) and the one just picked by $N^{(1)}$ as its most preferred node, and so on. Thus, there are $(x - 1)!$ ways to assign the most preferred node to each of the nodes in S_g. The assignment of the second preferred node for each node in S_g without violating the conditions of Theorems 1 and 2 can be obtained by shifting the assignment of the most preferred node as described above, i.e., $N^{(i)}$ picks $N^{(i+1)}$'s most preferred node for $i = 1, \ldots, x - 1$ while $N^{(x)}$ picks $N^{(1)}$'s most preferred node. The rest of the nodes can be ordered similarly. Thus, there are at least $(x - 1)!$ ways to modify a preferred list for the nodes in S_g without violating the conditions of Theorems 1 and 2. □

When N_i becomes faulty, the preferred list of each node in $S_{N_i}^{-1}$ needs to be adjusted. Since adjusting a preferred list will introduce computation and communication overheads, it is desirable to design an algorithm which requires minimal adjustments in case of node failures. Moreover, in the fault-tolerant backup queue approach, the most preferred node should be located as close to the failed node as possible to reduce the communication overhead for maintaining/updating the backup queue.

3.2 Minimizing the Number of Adjustments

An adjustment algorithm is said to be "optimal" if it requires a minimal number of nodes to be adjusted.

THEOREM 4. When N_i is faulty, the minimal adjustment to the preferred list of each node in $S_{N_i}^{-1}$ is to change the kth preferred node of either one node if $M_k(N_i) \neq M_k^{-1}(N_i)$, or two nodes if $M_k(N_i) = M_k^{-1}(N_i)$, $k = 1, \ldots, \sigma$ without violating the conditions of Theorems 1 and 2.

PROOF. Since N_i is the kth preferred node of $M_k^{-1}(N_i)$, it needs to be replaced by a fault-free node. Note that according to Theorem 1, the kth preferred node of N_i (i.e., $M_k(N_i)$) will not be selected as the kth preferred node by any other node. If $M_k(N_i) \neq M_k^{-1}(N_i)$, one can simply substitute $M_k(N_i)$ for N_i as the kth preferred node of $M_k^{-1}(N_i)$. This adjustment will satisfy the conditions of Theorems 1 and 2. However, in the case of $M_k(N_i) = M_k^{-1}(N_i)$, $\forall k$, the above adjustment is meaningless. We now show that there always exist a pair of nodes, say N_x and N_y, such that changing the kth preferred node of both $M_k(N_i)$ and N_x will always satisfy the conditions of Theorems 1 and 2.

For notational convenience, let N_x and N_y be two distinct nodes, such that $N_y = M_k(N_x)$, $M_k^{-1}(N_i) \notin S_{N_x}$, and $N_y \notin S_{M_k^{-1}(N_i)}$, i.e., $M_k^{-1}(N_i)$ is not in N_x's buddy set and N_y is not in $M_k^{-1}(N_i)$'s buddy set. Then we can satisfy the conditions of Theorem 1 and 2 with the following adjustments:

1) replace N_i by N_y as the the kth preferred node of $M_k^{-1}(N_i)$, and

2) replace N_y by $M_k^{-1}(N_i)$ as the the kth preferred node of N_x.

Theorem 1 is satisfied by assuring $M_k^{-1}(N_i) \notin S_{N_x}$, and $N_y \notin S_{M_k^{-1}(N_i)}$. Before making this adjustment, $M_k^{-1}(N_i)$ is selected as the kth preferred node by N_i only and it will not be selected as the kth preferred node of any other node, when N_i becomes faulty. So, after making the adjustment, $M_k^{-1}(N_i)$ will be selected as the kth preferred node by node N_x only, and N_y will be selected as the kth preferred node by $M_k^{-1}(N_i)$, thus satisfying the conditions of Theorem 2.

As long as the size of a buddy set satisfies the conditions of Theorem 2, there will always exist the (N_x, N_y) pair in a Q_4 or larger hypercube. The restriction $N_y \notin S_{M_k^{-1}(N_i)}$ implies that N_y must be at least two hops away from $M_k^{-1}(N_i)$, and the relation $N_y = M_k(N_x)$ implies that N_x must be at least three hops away from $M_k^{-1}(N_i)$. (This modification is also shown in Table 2.) □

TABLE 2
ILLUSTRATION OF MODIFICATION OF A PREFERRED LIST

In the original preferred list	
Nodes	kth preferred node
N_i (faulty)	$M_k(N_i)$
$M_k^{-1}(N_i) = M_k(N_i)$	N_i (faulty)
N_x	$N_y = M_k(N_x)$
Modification	
$M_k(N_i)$	N_y
N_x	$M_k(N_i)$

We can find the (N_x, N_y) pair systematically for each node in $S_{N_i}^{-1}$ as follows. Since N_y will replace N_i as the kth preferred node of $M_k^{-1}(N_i)$, N_y must not be a node in $M_k^{-1}(N_i)$'s buddy set. The search can follow the sequence of nodes in $M_k^{-1}(N_i)$'s preferred list, starting from the first node outside of its buddy set. This node can be found by $M_k^{-1}(N_i) \oplus I_p \oplus I_q$ if it is two hops away from $M_k^{-1}(N_i)$, or by $M_k^{-1}(N_i) \oplus I_p \oplus I_q \oplus I_r$ if it is three hops away, where $p = 1, 2, \ldots, n - 1, 0$, $q = p + 1, 2, \ldots, n - 1$, and $r = q + 1, 2, \ldots, n - 1$. The next step is to check if $M_k(N_i)$ is not in N_x's buddy set. This can be guaranteed if N_x is located farther away from $M_k(N_i)$. Without loss of generality, one can consider the case of $k \leq n$, and assume that the buddy set of a node N_k contains all nodes one hop away from N_k and some of the nodes which are two hops away from N_k. The faulty nodes can be replaced by the following algorithm.

Preferred List Adjustment Algorithm 1

```
for $k = 1$ to $\sigma$ do
    1) Find $N_y = M_k^{-1}(N_i) \oplus I_p \oplus I_q$, such that $p, q \neq k - 1$ and $N_y \notin S_{M_k^{-1}(N_i)}$
    2) Replace $N_i$ by $N_y$
    3) Find $N_x = N_y \oplus I_{k-1}$
    4) Replace $N_y$ by $M_k^{-1}(N_i)$
```

To illustrate the above adjustment algorithm, consider a Q_8 as an example. Without loss of generality, assume the buddy set size is 10 [6] and let N_0 be the faulty node. According to the definition of the preferred list, $S_{N_0} = \{N_1, N_2, N_4, N_8, N_{16}, N_{32}, N_{64}, N_{128}, N_6, N_{10}\}$. Since N_1 selects N_0 as its most preferred node, we must find a node to replace N_0. According to Algorithm 1, the search for the pair (N_x, N_y) starts from the first node outside N_1's buddy set. So, $N_y = M_1^{-1}(N_0) \oplus I_1 \oplus I_4 = N_1 \oplus I_1 \oplus I_4 = N_{19}$ and $N_x = N_{18}$. The new most preferred node of N_1 will be N_{19} and the new most preferred node of N_{18} will be N_1. Similarly, the N_2's buddy set = $\{N_3, N_0, N_6, N_{10}, N_{18}, N_{34}, N_{66}, N_{130}, N_4, N_8\}$. The first node outside N_2's buddy set is $N_y = M_2^{-1}(N_0) \oplus I_1 \oplus I_4 = N_2 \oplus I_1 \oplus I_4 = N_{16}$ and $N_x = N_{18}$. So, the new second preferred node of N_2 (N_{18}) will be N_{16} (N_2).

THEOREM 5. *The preferred list resulting from Algorithm 1 will satisfy the conditions of Theorems 1 and 2 in the presence of faulty nodes.*

PROOF. According to the definition of a preferred list, $M_k(N_i) = N_i \oplus I_{k-1}$ and $N_y = M_k^{-1}(N_i) \oplus I_p \oplus I_q = N_i \oplus I_{k-1} \oplus I_p \oplus I_q$. Then $N_x = M_k^{-1}(N_y) = N_y \oplus I_{k-1} = N_i \oplus I_p \oplus I_q$ is farther away from $M_k^{-1}(N_i)$ if $p, q \neq k - 1$. Since a preferred list is generated according to the distance between nodes, if N_y is not in $M_k^{-1}(N_i)$'s buddy set, then N_x, which is farther away from $M_k^{-1}(N_i)$, will also not be in its buddy set. Thus, the (N_x, N_y) pair will satisfy the conditions of Theorems 1 and 2. □

3.2.1 *Reduction of Average Internode Distance in a Buddy Set*

Although Algorithm 1 can modify the preferred lists with a minimal number of adjustments, the distance between a node and the nodes in its buddy set is found to increase significantly after making these adjustments. Whenever a node N_i becomes faulty, the nodes in $S_{N_i}^{-1}$ need to adjust their preferred lists. In each of these adjustments a pair of nodes, N_x and N_y, need to be selected. According to Algorithm 1 every node in $S_{N_i}^{-1}$ will select the first fault-free node as N_y in $\overline{S_{M_k^{-1}(N_i)}}$. Suppose $N_y = N_i \oplus I_{k-1} \oplus I_p \oplus I_q$, then $N_x = M_k^{-1}(N_y) = N_y \oplus I_{k-1} = N_i \oplus I_p \oplus I_q$. So, in each of these adjustments $N_x = N_i \oplus I_p \oplus I_q$ is the same node. In other words, after completing the adjustment for every node in $S_{N_i}^{-1}$, the nodes in $N_i \oplus I_p \oplus I_q$'s buddy set will be at least three hops away from this node, making the average distance greater than 2, while the average distance is around 1 in all other nodes. Consider the previous example Q_8 and assume N_0 is faulty. After completing the adjustments for all nodes in N_0's buddy set, N_{18}'s buddy set will become $\{N_1, N_2, N_4, N_8, N_{16}, N_{32}, N_{64}, N_{128}, N_6, N_{10}\}$.

To alleviate this problem, Algorithm 1 is modified in such a way that a different pair of nodes, N_x and N_y, are selected for each node in $S_{N_i}^{-1}$. In such a case, after completing the adjustment for every node in $S_{N_i}^{-1}$ the average distance between a node and the nodes in its buddy set will become smaller than the distance resulting from Algorithm 1. From the definition of a preferred list, the nodes which are two hops away from N_i are ordered according to the operation of $N_i \oplus I_j \oplus I_k$ ($j = 1, \ldots, n - 1, 0$, and $j + 1 \leq k \leq n - 1$). For convenience, the nodes with the same j in the above operation are grouped as parcel j, denoted as $\Psi_j(i)$ (parcel j of node i). The kth node in $\Psi_j(i)$ can be obtained by $N_i \oplus I_j \oplus I_k$.

Preferred List Adjustment Algorithm 2

```
For $k = 1$ to $\sigma$ do
    1) Find the first parcel $j$ of $M_k^{-1}(N_i)$ for $j \neq k - 1$.
    2) if $j = 1$ then
            if $j + k \leq n - 1$ then $N_y = M_k^{-1}(N_i) \oplus I_j \oplus I_{j+k}$
            else $j \leftarrow j + 1$ goto step 2.
       else $p = k - \sum_{\ell=1}^{\ell=j} |\Psi_\ell(i)|$
            if $p \geq 0$ then $N_y = M_k^{-1}(N_i) \oplus I_j \oplus I_{j+p}$
            else $N_y = M_k^{-1}(N_i) \oplus I_j \oplus I_{j+1}$
    3) Replace $N_i$ by $N_y$ as $M_k^{-1}(N_i)$'s $k$th preferred node.
    4) Find $N_x = N_y \oplus I_k$.
    5) Replace $N_y$ by $M_k(N_i)$ as $N_x$'s $k$th preferred node.
end_do
```

3.3 Minimization of Average Internode Distance in a Buddy Set

Although the number of adjustments in each preferred list is minimized by Algorithm 2, the distance between a node and its most preferred node is not necessarily minimal. According to Algorithm 2, the kth preferred node of $M_k^{-1}(N_i)$ (i.e., N_i) is replaced by N_y which is at least two hops away from $M_k^{-1}(N_i)$. Moreover, the kth preferred node of N_x (i.e., N_y) is replaced by $M_k(N_i)$ which is at least three hops away from N_x. In the case of $k = 1$ the most preferred nodes of $M_1^{-1}(N_i)$ and N_x will be two and three hops away from $M_1^{-1}(N_i)$, respectively.

Since each node needs to be aware of the tasks arriving at its most preferred node and vice versa, the longer distance between them means the larger communication delay. An alternative approach is to minimize the distance between the nodes and their replacement nodes. The following algorithm will adjust the preferred list of each fault-free node such that the most preferred node is located one or two hops away from each fault-free node.

Preferred List Adjustment Algorithm 3

```
For $k = 1$ do
    if $M_k^{-1}(N_i)$ is not faulty then
        Replace $N_i$ by $M_{n-k}(M_k^{-1}(N_i))$ as the $k$th preferred node of $M_k^{-1}(N_i)$
        Replace $M_{n-k}(M_k^{-1}(N_i))$ by $M_k(N_i)$ as the $k$th preferred node of $M_k^{-1}(M_{n-k}(M_k^{-1}(N_i)))$
        Use the same procedure in Algorithm 2 to find a node to replace
        $M_k^{-1}(M_{n-k}(M_k^{-1}(N_i)))$ as the $(n - k)$th preferred node of $M_k^{-1}(N_i)$
    else    no adjustment
```

TABLE 3
COMPARISON OF PREFERRED LIST ADJUSTMENT ALGORITHMS IN AN 8-CUBE SYSTEM WITH 15 FAULTY NODES

	Average distance		Distance to most preferred node		Number of adjustments on a node
	Lower	Upper	Lower	Upper	
Algorithm 1	1.33	2.89	1	4	2
Algorithm 2	1.33	1.46	1	4	2
Algorithm 3	1.33	1.51	1	2	4

```
end_do
for $k = 2$ to $\sigma$ do same as Algorithm 2 end_do
```

Algorithm 3 can be explained as follows. In the first step $M_1^{-1}(N_i)$ chooses N_i as its most preferred node, so $M_{n-1}(M_1^{-1}(N_i))$ is selected to replace N_i. In a Q_n, the first nth preferred nodes in a buddy set are within one hop. So, the distance between $M_1^{-1}(N_i)$ and $M_{n-1}(M_1^{-1}(N_i))$ is one hop. However, $M_{n-1}(M_1^{-1}(N_i))$ is originally selected by another node $(M_1^{-1}(M_{n-1}(M_1^{-1}(N_i)))$ as the most preferred node, so $M_k(N_i)$ needs to replace $M_{n-1}(M_1^{-1}(N_i))$ and the distance between $M_k(N_i)$ and $M_1^{-1}(M_{n-1}(M_1^{-1}(N_i))$ is two hops. The other two adjustments are to find a replacement for $M_{n-1}(M_1^{-1}(N_i))$ as the $(n - 1)$th preferred node for $M_1^{-1}(N_i)$, so the total number of adjustments will be four when $k = 1$. For the cases when $k > 1$, only two adjustments are needed as done in Algorithm 2.

Note that in Algorithm 3, the distance between a node and its most preferred node is less than two as long as the node is connected to the system, but it requires four adjustments (if $M_1^{-1}(N_i)$ is not faulty). This algorithm can be run in parallel on all nodes in $S_{N_i}^{-1}$, as long as N_i's failure is detected by the nodes in $S_{N_i}^{-1}$. When there are more than one faulty node, Algorithms 2 and 3 can be used sequentially to make the adjustments.

The number of adjustments and the average distance between a node and its preferred node resulting from these algorithms are compared in Table 3. It is shown that Algorithm 1 results in the largest distance while Algorithm 3 results in minimal distance between a node and its preferred node. If the number of adjustments is the main concern, Algorithm 2 should be used because it results in a smaller average distance than Algorithm 1 while minimizing the number of adjustments.

4 IMPLEMENTATION AND ANALYSIS OF FAULT-TOLERANT BACKUP QUEUE

Based on the proposed adjustment algorithms, a simple fault-tolerant mechanism can be implemented to reduce the number of task losses when a node fails. Each node maintains two (or more) task queues, one for its own arrivals (EAQ) and the other for the arrivals from its most preferred node (BKQ). The BKQ is updated upon arrival/completion of each task at a node¢s most preferred node.

Upon N_i's failure, $M_1(N_i)$ will accept all the tasks in its BKQ (as bursty arrivals). $M_1(N_i)$ will process all of these tasks if it is underloaded; otherwise, it will transfer some or all of these tasks to the underloaded nodes in its buddy set. The most important issue in this approach is to adjust the preferred list of $M_1^{-1}(N_i)$ and update its BKQ with the newly assigned most preferred node.

Since each node in $S_{N_i}^{-1}$ can execute the adjustment algorithm concurrently with other nodes in this set, the communication delay, T_{adjust}, associated with updating the BKQ of $M_1^{-1}(N_i)$ and the newly assigned node N_x, can be derived as $T_{adjust} = (k_1 + k_2) \times T_t + T_c$, where k_1 and k_2 is the number of tasks in the EAQ of M-11(Ni) and $M_1(N_b)$, respectively, T_t is the task transfer time, and T_c is the communication time between $M_1(N_i)$ and N_b, or between $M_1^{-1}(N_b)$ and $M_1(N_i)$ required to set up the updating procedure. If $M_1(N_i)$ fails before completing the adjustment of preferred lists and the updating procedure, the tasks in EAQ and BKQ in this node will be lost. Since the communication delay for updating BKQ increases with the number of preferred lists to be adjusted, it is important to use an algorithm that requires minimal adjustment.

To further reduce task losses, multiple BKQs can be provided to maintain/update the tasks from a node's second, or higher, preferred nodes. But the communication overhead and delay for maintaining/updating these BKQs in each node may become high, thus offsetting any improvement to be gained by using multiple BKQs. In fact, our simulation results show that the improvement of having more than two BKQs in each node is insignificant, as compared to a single BKQ when the number of faulty nodes is less than 25% of the total number of nodes in the system.

4.1 Notation

- MTBF : mean time between failure $(1/\lambda)$.
- T_{exe} : (average) task execution time.
- α : ratio of mean time between failure (MTBF) to T_{exe}.
- β : ratio of task transfer time to T_{exe}.
- A_T : average number of tasks queued in a node.
- P_f: probability of a node failure before completing the updating process.
- $P_f(t_1, t_2)$: probability of a node failure in time interval $[t_1, t_2]$.
- t_u : time to transfer a task between two adjacent nodes.
- $\lambda e^{-\lambda t}$: probability density function of a node failure in $[0, t]$.
- T_{lost} : average number of lost tasks in a node.
- $T_e^{N_i}$: N_i's queue for externally arriving tasks.
- $B_j^{N_i}$: the jth backup queue of node N_i.

4.2 Single Backup Queue

Suppose N_1 and N_2 back up each other, then $B_1^{N_1} = T_e^{N_2}$ and $B_1^{N_2} = T_e^{N_1}$. If N_1 is faulty, N_2 will take over the tasks in its backup queue as its own tasks, the new task queue of N_2 will be $T_e^{N_2} \cup T_e^{N_1}$. Since N_2 was backed up by N_1, it must find a new backup node and transfer its $T_e^{N_2}$ to the newly-selected node. If this process is completed before N_2 becomes faulty, no tasks will be lost; otherwise, some of the tasks will be lost. Let the time of N_1's failure be the reference time t_0. Since the number of tasks in $T_e^{N_2}$ is $2A_T$ after N_1 becomes faulty, $2A_T$ tasks will be lost if N_2 fails between $[t_0, t_u]$, $2A_T - 1$ tasks will be lost if N_2 fails between $[t_u, 2t_u]$, and one task will be lost if N_2 fails between $[(2A_T - 1)t_u, 2A_Tt_u]$. No task will be lost in N_2 after $2A_Tt_u$, because all of these tasks will be backed up by another node. Thus, the average number of tasks lost is the sum of the product of the number of tasks lost and the probability of N_2 fails in each interval.

$$T_{lost} = 2A_T\, P_f(t_0, t_u) + \sum_{i=1}^{2A_T-1} (2A_T - i)\left[1 - P_f(t_0, it_u)\right] P_f(it_u, (i+1)t_u) \quad (4.1)$$

From the definition of $P_f(t_1, t_2)$, we have

$$P_f(t_1, t_2) = \int_{t_1}^{t_2} \lambda\, e^{-\lambda t} e^{-\lambda t_1} - e^{-\lambda t_2} \approx \lambda(t_2 - t_1)$$

when $\lambda\, t_1, \lambda\, t_2 \ll 1$. Then, we have $P_f(t_0, mt_u) = m\, \lambda t_u$ and $P_f(mt_u, (m+1)t_u) = \lambda t_u$, when $m\lambda t_u \ll 1$ and $\lambda t_u \ll 1$. Using the above equation, we can rewrite (4.1) as follows:

$$\begin{aligned} T_{lost} &= \lambda t_u \left[\sum_{i=1}^{2A_T} i - \lambda t_u \sum_{i=1}^{2A_T-1} (2A_T - i)\, i \right] \\ &= \lambda t_u \left[A_T(2A_T + 1) - \lambda t_u \frac{A_T(2A_T - 1)(2A_t + 1)}{3} \right] \\ &\approx \left[A_T(2A_T + 1) \right] \lambda t_u \ \text{ when } \lambda t_u \ll 1 \qquad (4.2) \end{aligned}$$

4.3 Double Backup Queues

Let N_1, N_2, and N_3 back up each other, then $B_1^{N_1} = T_e^{N_2}$, $B_2^{N_1} = T_e^{N_3}$, $B_1^{N_2} = T_e^{N_1}$, and $B_2^{N_3} = T_e^{N_1}$. Note that the first and second backup queue of N_3 and N_2 will be the task queue of some other nodes, respectively. If N_1 is faulty, N_2 will take over the tasks in its backup queue as its own tasks, the new task queue of N_2 will be $T_e^{N_2} \cup T_e^{N_1}$. Since N_2's first backup queue was in N_1, it must find a new first backup node and transfer its $T_e^{N_2}$ to the newly-selected node. If N_2 failed before completing this process, N_3 will take over its $B_2^{N_3}$, and the same process will start on N_3. However, if N_3 failed before completing this process, some of tasks in N_1 will be lost. The probability of losing a task is analyzed as follows. Suppose N_2 fails in $[0, t_u]$, then

$$\begin{aligned} T_{lost}^{(1)} &= P_f(0, t_u) \sum_{j=0}^{A_T-1} (A_T - j)\left[1 - P_f(0, jt_u)\right] P_f(jt_u, (j+1)t_u) \\ &= (\lambda t_u)^2 \sum_{j=0}^{A_T-1} (A_T - j)(1 - j\lambda t_u). \end{aligned}$$

Suppose N_2 fails in $[t_u, 2t_u]$, then

$$\begin{aligned} T_{lost}^{(2)} &= (1 - P_f(0, t_u)) P_f(t_u, 2t_u) \sum_{j=0}^{A_T-2} (A_t - j - 1)\left[1 - P_f(0, jt_u)\right] P_f(jt_u, (j+1)t_u) \\ &= (1 - \lambda t_u)(\lambda t_u)^2 \sum_{j=0}^{A_T-2} (A_T - j - 1)(1 - j\lambda t_u). \end{aligned}$$

Since N_2 can fail at any time in $[0, A_T t_u]$, we have

$$\begin{aligned} T_{lost} &= \sum_{i=0}^{A_T-1} (1 - P_f(0, it_u)) P_f(it_u, (i+1)t_u) \\ &\quad \sum_{j=0}^{A_T-i-1} (A_T - i - j)\left[1 - P_f(0, jt_u)\right] P_f(jt_u, (j+1)t_u) \\ &\approx (\lambda t_u)^2 \left[\sum_{i=0}^{A_T-1} i^2 + (\frac{1}{2} - 2\, A_T)i + A_T^2 - \frac{A_T - 1}{2} \right] \\ &= (\lambda t_u)^2 A_T \left(\frac{1}{3} A_T^2 + \frac{A_T}{4} + \frac{5}{12} \right) \qquad (4.3) \end{aligned}$$

Equation (4.3) gives the probability of task loss when N_1's failure is followed by N_2 and N_3. Since N_2 and N_3 both have one backup queue on N_1, if the nodes that hold another backup queue of N_2 or N_3 fail, the task in N_2 and N_3 will be lost. This probability can be expressed exactly as (4.3). So, the total probability of task loss is three times that of (4.3). However, there are many higher-order probabilities of task loss. That is, when N_1, N_2, N_3, N_4, and N_5 all failed, if the nodes that hold N_4 and N_5's backup queue fail, the tasks in N_4 and N_5 will also be lost, and so on. Since there are many combinations (2^N) that will result in a higher-order probability of losing a task and these probabilities decrease exponentially as the number of nodes involved increases, the combinations with more than five nodes are ignored in the analysis.

4.4 Triple Backup Queues

In the case of three backup queues and let N_1, N_2, N_3, and N_4 back up each other, we have $B_1^{N_1} = T_e^{N_2}$, $B_2^{N_1} = T_e^{N_3}$, $B_3^{N_1} = T_e^{N_4}$, $B_1^{N_2} = T_e^{N_1}$, $B_2^{N_3} = T_e^{N_1}$, and $B_3^{N_4} = T_e^{N_1}$, where the first, second, and third backup queue of N_4, N_3, and N_2 will be the task queue of some other nodes, respectively. The probability of task loss can be expressed as follows. (Due to its complexity, a closed-form solution cannot be found.)

$$\begin{aligned} T_{lost} &= \sum_{i=0}^{A_T-1} (1 - P_f(0, it_u)) P_f(it_u, (i+1)t_u) \sum_{j=0}^{A_T-i-1} (A_T - i - j)\left[1 - P_f(0, j\, t_u)\right] \\ &\quad P_f(jt_u, (j+1)t_u) \sum_{k=0}^{A_T-i-j-1} (A_T - i - j - k)\left[1 - P_f(0, kt_u)\right] P_f(kt_u, (k+1)t_u) \\ &= \sum_{i=0}^{A_T-1} (1 - i\lambda t_u)\lambda t_u \sum_{j=0}^{A_T-i-1} (A_T - i - j)(1 - j\lambda t_u)\lambda t_u \\ &\quad \sum_{k=0}^{A_T-i-j-1} (A_T - i - j - k)(1 - k\lambda t_u)\, \lambda t_u\, . \end{aligned}$$

For the case of multiple backup queues, one can express the probability of task loss similarly to the above equation, but it is too difficult to derive a closed-form solution. Note that λt_u in all equations is equal to α/β.

4.5 Simulation Results

In addition to modeling, the performance of the proposed adjustment algorithms and fault-tolerant BKQ mechanisms is also evaluated via simulations. The results in [6] show that threshold pattern $TH_u = 1$, $TH_f = 2$, $TH_v = 3$ performs well in the capability of load sharing for a wide range of system load, and thus, is used in the simulations. The size of a buddy set is chosen to be 10, because the performance improvement beyond this size is shown to be insignificant [6]. The system load is varied from 0.5 (medium-loaded) to 0.9 (overloaded) and the number of faulty nodes is changed from 5% to 50% of the total number of nodes in an 8-cube system.

The first simulation is run without adjusting preferred lists. Faulty nodes are randomly generated before the simulation and no new faults are assumed to occur during the simulation. Since the faulty nodes are simply dropped from the preferred lists, the *missing probability* (or probability of missing a task's deadline) increases very fast with the number of node failures when preferred lists are not adjusted. The missing probabilities in the presence of faulty nodes are normalized to the case without faulty nodes in Fig. 2. In case of 50% faulty nodes and system load at 0.8, the missing probability can be two times as high as the case without faulty nodes.

Another simulation is run to test the goodness of the proposed adjustment algorithm. In order to eliminate other factors that may influence the results, the faulty nodes are randomly generated, and the preferred lists of the nodes with faulty nodes in the buddy set are adjusted before the simulation. No new faults are assumed to occur throughout the simulation. These results are superimposed in Fig. 2. Surprisingly, the missing probability for the case with faulty nodes is found to be nearly the same as that for the case without faulty nodes regardless of the fraction of faulty nodes and system load.

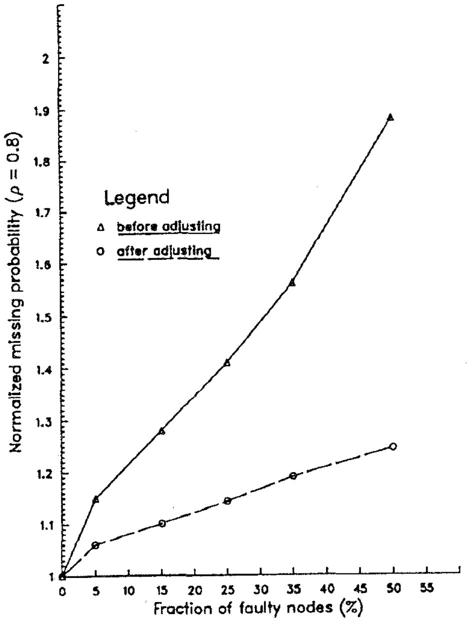

Fig. 2. Comparison of missing probabilities.

The performance of the fault-tolerant BKQ method is measured by the number of lost tasks. The tasks in a node and its most preferred node will be lost when these two nodes fail within the time period, T_{adjust}, required to adjust the preferred lists. The simulation results are tabulated in Table 4, where three BKQs (= 1, 2, 3) along with the case of BKQ = 0 are listed at each run under different system loads. The analytical results are shown in Figs. 3 and 4. The simple fault-tolerant mechanism with BKQ = 1 is shown to completely eliminate the number of lost tasks when the number of faulty nodes is less than 15% and the system load is less than 0.7. Except in an extreme case when the number of faulty nodes reaches 50% of the total number of nodes, this simple approach can reduce the number of lost tasks significantly, as compared to the approach without BKQ.

Using multiple BKQs reduces further the number of lost tasks for the cases of a higher percentage of faulty nodes. However, due to the increased communication overhead and delay with the multiple BKQs, the number of lost tasks cannot be completely eliminated in the case of 50% or higher percentage of faulty nodes. However, as shown in Fig. 3, the cases of BKQ = 2 and 3 yield higher tasks loss as compared to BKQ = 1 when the node is not too reliable ($MTBF = 200T_{exe}$). On the other hand, when the node is relatively reliable ($MTBF = 5{,}000T_{exe}$) the case of using multiple backup queue did reduce the number of task losses except when $\beta > 0.7$ for the case of two backup queues.

The analytical results agree well with the simulation results in all cases. For example, consider the case of 35% faulty node, $\beta = 0.1$, and $\rho = 0.8$. The A_T is approximately equal to 1.5 [6] (Table 1) and MTBF is close to 5,000 times T_{exe}. From Fig. 4, the average number of tasks lost on a node resulting from using one, two, and three backup queues are 8×10^{-5}, 1.0×10^{-5}, and 1.1×10^{-7}, respectively. The total number of lost tasks is equal to the above value times the total number of processed tasks which is around 4.1×10^{4} in the simulation. So, the total number of lost tasks for one, two, and three backup queues will be 32.8, 4.1, and 0.045, while the simulation results are 27, 6, and 0, respectively.

An interesting problem found during the simulation is that the probability of missing task deadlines in the case of using fault-tolerant BKQs is higher than the case without any BKQ. The reason is that a node may fail during the processing of a task, and this task is restarted on another node instead of continuing its execution from the point in time when the node failed. Although this task can be successfully completed by another node, the total processing time may often exceed its deadline if it is queued at the new node before its execution.

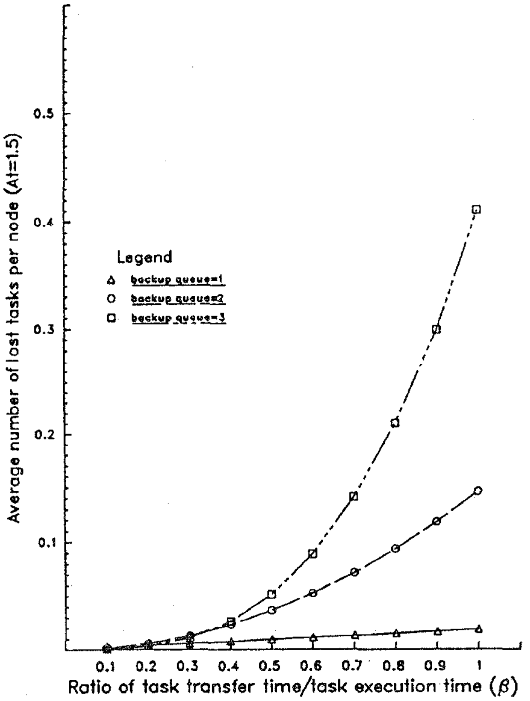

Fig. 3. Number of lost tasks vs. β ($MTBF = 200 \times T_{exe}$).

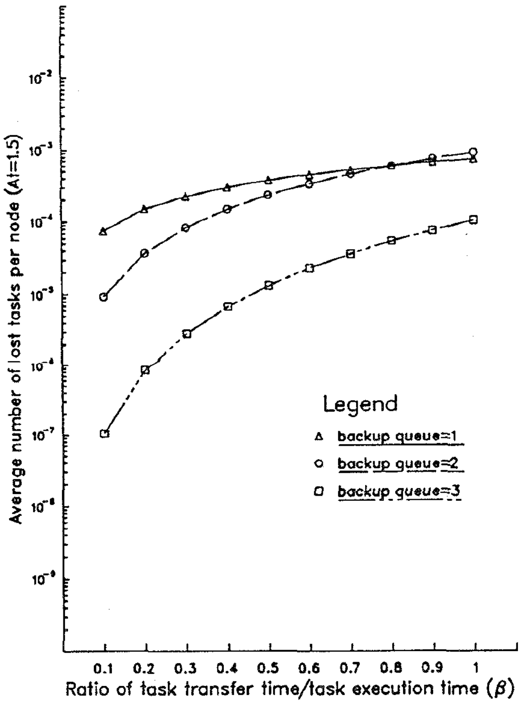

Fig. 4. Number of lost tasks vs. β ($MTBF = 5{,}000 \times T_{exe}$).

TABLE 4
THE NUMBER OF LOST TASKS VS. NUMBER OF BACKUP QUEUES

% of faulty nodes		5%	15%	25%	35%	50%
System load	# of BKQs					
0.5	0	10	29	36	50	67
	1	0	1	3	8	29
	2	0	0	0	2	10
	3	0	0	0	0	2
0.6	0	14	35	46	74	106
	1	0	3	4	10	35
	2	0	0	0	5	13
	3	0	0	0	0	3
0.7	0	23	46	73	105	180
	1	0	3	4	15	47
	2	0	0	0	5	15
	3	0	0	0	0	5
0.8	0	31	59	87	134	211
	1	0	3	6	27	61
	2	0	0	1	6	26
	3	0	0	0	0	9
0.9	0	42	86	96	164	297
	1	0	5	11	35	84
	2	0	0	5	7	30
	3	0	0	0	1	12

5 CONCLUSION

Several algorithms to adjust preferred lists and implement a fault-tolerant mechanism are proposed and evaluated in this paper. The preferred lists modified by the proposed algorithms are shown to retain their original properties—thus solving both the coordination and congestion problems—regardless of the number of faulty nodes in the system. Moreover, these algorithms can either minimize the number of adjustments or minimize the distance between a node and the node in its buddy set. A simple fault-tolerant BKQ is implemented based on the proposed algorithms. The communication overhead and delay for maintaining/updating the BKQ is shown to be minimal, thus reducing the number of task losses.

There remain several issues worth further investigation. First, the preferred lists in the buddy sets are generated according to the physical distance between nodes, and the nodes with shorter distance between them will receive higher preference. However, when some nodes failed, the distance between nodes might be changed. How to modify the preferred lists to adapt to this change needs to be studied further. Second, the missing probability in the case with a BKQ is found to be higher than the case without a BKQ. In other words, although some tasks are saved by using the fault-tolerant BKQ, its completion time often exceeds the deadline. How to design a fault-tolerant BKQ for real-time applications is an interesting problem.

ACKNOWLEDGMENTS

The work reported in this paper was supported in part by the U.S. Office of Naval Research under Grant N00014-94-1-0229, and the U.S. National Science Foundation under grant MIP–9203895. Any opinions, findings, and recommendations expressed in this publication are those of the authors and do not necessarily reflect the view of the funding agencies.

This paper is a revised version of "Load Sharing in Hypercube Multicomputers in the Presence of Node Failures," by Y.-C. Chang and K.G. Shin, which appeared in the *Proceedings of the 21st IEEE Fault-Tolerant Computing Symposium*, Montreal, Canada, June 1991, pp. 188–195.

REFERENCES

[1] K.G. Shin and Y.-C. Chang, "Coordinated Load Sharing in Hypercube Multicomputers," *IEEE Trans. Computers*, vol. 44, no. 5, pp. 669-682, May 1995.

[2] Y.-T. Wang and R.J.T. Morris, "Load Sharing in Distributed Systems," *IEEE Trans. Computers*, vol. 34, no. 3, pp. 204-217, Mar. 1985.

[3] D.L. Eager, E.D. Lazowska, and J. Zahorjan, "Adaptive Load Sharing in Homogeneous Distributed Systems," *IEEE Trans. Software Eng.*, vol. 12, no. 5, pp. 662-675, May 1986.

[4] L.M. Ni, C.W. Xu, and T.B. Gendreau, "A Distributed Drafting Algorithm for Load Balancing," *IEEE Trans. Software Eng.*, vol. 11, no. 10, pp. 1,153-1,161, Oct. 1985.

[5] T.L. Casavant, "Analysis of Three Dynamic Distributed Load-Balancing Strategies with Varying Global Information Requirements," *Proc. Seventh IEEE Int'l Conf. Distributed Computing Systems*, pp. 185-192, 1987.

[6] K.G. Shin and Y.-C. Chang, "Load Sharing in Distributed Real-Time Systems with State Change Broadcasts," *IEEE Trans. Computers*, vol. 38, no. 8, pp. 1,124-1,142, Aug. 1989.